

**« VIRTUAL-VERTIGO : APPRÉHENDER LE VERTIGE
GRÂCE A LA RÉALITE VIRTUELLE »**



Thèse de Bachelor présentée par

Madame Ana Sofia ANTÓNIO DOMINGOS

pour l'obtention du titre Bachelor of Science HES-SO en

**Ingénierie des technologies de l'information avec orientation en
Logiciels et systèmes complexes**

Septembre 2015

Professeur HES responsable TB

Paul ALBUQUERQUE

INGÉNIERIE DES TECHNOLOGIES DE L'INFORMATION

ORIENTATION – LOGICIELS ET SYSTEMES COMPLEXES

VIRTUAL VERTIGO : APPRÉHENDER LE VERTIGE GRÂCE À LA RÉALITÉ VIRTUELLE

Descriptif :

Un dispositif de type Google Cardboard permet de transformer un smartphone en un casque de réalité augmentée via deux lentilles et un montage en carton. Le faible coût du dispositif technique (quelques francs hors smartphone) laisse imaginer des applications pour un large public. Le monde virtuel en trois dimensions sera affiché sur le smartphone avec la technologie web X3DOM. Un utilisateur devra pouvoir s'immerger à travers ce dispositif dans un monde virtuel consistant en une planche au-dessus du vide entre deux immeubles. La Kinect V1 ou V2 de Windows permettra de récupérer les mouvements et les déplacements de la personne sur la planche afin de positionner le personnage représentant l'utilisateur dans le monde virtuel.

La Kinect est un appareil de détection de mouvements, initialement associée à la console de jeu Xbox. La Kinect sera branchée sur un ordinateur qui fera office de serveur pour le smartphone et qui enverra les données récupérées. En outre, il faudra aussi ajouter d'autres types de sensations (sonores ou sensorielles) pour que l'immersion de l'utilisateur dans le monde virtuel soit la plus efficace possible.

**Travail demandé :**

Dans les limites du temps à disposition, la candidate effectuera les tâches suivantes :

1. Prise en main des technologies X3DOM et NodeJS, test des Google Cardboard
2. Génération du décor 3D p.ex. avec Blender
3. Simulation d'un déplacement fictif dans ce décor 3D
4. Prise en main de la Kinect
5. Récupération des mouvements et déplacements d'une personne, puis simulation
6. Intégration de ces données dans le décor 3D en mode offline
7. Création d'un personnage virtuel représentant l'utilisateur
8. Détection des mouvements de la tête avec les capteurs du smartphone et repositionnement de la vue
9. Mise en place de l'architecture complète

Candidate :

Mme. DOMINGOS Ana Sofia

Filière d'études : ITI

Professeur responsable :

ALBUQUERQUE Paul

En collaboration avec : Jérémie GOBET
Travail de bachelor soumis à une convention
de stage en entreprise : **non**
Travail de bachelor soumis à un contrat de
confidentialité : **non**

Timbre de la direction



Résumé :

La réalité virtuelle est une simulation informatique interactive immersive, en temps réel. La simulation peut être visuelle, sonore ou haptique d'environnements réels ou imaginaires. L'immersion est réalisée à l'aide de dispositifs. La réalité virtuelle peut être utilisée pour des *Serious Games* pour aider une personne à combattre une peur en la mettant en situation.

Le projet *Virtual-Vertigo* est un *Serious Game* visant à aider les personnes ayant le vertige en utilisant la réalité virtuelle. Le vertige est un trouble affectant une personne dans le contrôle de sa situation dans l'espace.

L'objectif de *Virtual-Vertigo* est de réaliser un exercice permettant de combattre le vertige en utilisant un casque de réalité virtuelle constitué d'un smartphone, de lentilles et de carton.

L'exercice mis en place consiste à mettre une personne en situation de stress sur une planche reliant deux immeubles qu'il doit traverser.

Le projet *Virtual-Vertigo* est une application Web utilisant le *Framework X3DOM* permettant d'afficher des scènes 3D. Deux scènes 3D d'une planche reliant deux immeubles ont été modélisées. Ces scènes sont transformées de façon stéréoscopique afin de pouvoir être utilisées sur les *Google CardBoard* servant de casque de réalité virtuelle. Un *Kinect version 1* capture les mouvements de la personne en simulation. Le *Kinect* transmet ses données au *serveur Web Virtual-Vertigo*. Ce serveur Web sert de pont entre le *Kinect* et les navigateurs (*smartphone* et PC). Il retransmet les positions des articulations de la personne que le *Kinect* capture aux clients. Ce projet contient également des solutions d'immersions telles qu'une planche placée au sol et un bruit ambiant.



Le travail restant est la correction des problèmes persistants : le décalage de la vue lorsque le *smartphone* prend la main, la pose des textures lors de la rotation de la vue sur le *smartphone* et la visualisation non optimale des membres lors la simulation.

Candidate :

Professeur(s) responsable(s) :

Timbre de la direction

Mme. DOMINGOS ANA SOFIA Albuquerque Paul

Filière d'études : ITI

En collaboration avec : Jérémie Gobet
Travail de bachelor soumis à une convention
de stage en entreprise : non
Travail de bachelor soumis à un contrat de
confidentialité : non

--

Avant propos

Ce rapport décrit mon travail de Bachelor, réalisé sur une période de 12 semaines au sein de hebia. Le sujet de ce travail de Bachelor, a été proposé par M. JÉRÉMY GOBET assistant à hebia à la filière ITI est l'implémentation du projet *Virtual-Vertigo*. Ce projet consiste à mettre en place un exercice pour aider les personnes à combattre la peur du vide en utilisant la réalité virtuelle avec des casques de réalité virtuelle en carton et un smartphone. A noter que ce travail est la continuité de mon projet de semestre qui consistait à la prise en main, à la recherche et au choix des technologies et outils du projet *Virtual-Vertigo*.

Impressions personnelles

Ce travail a été très intéressant, amusant et attractif à mettre en place. En effet, ce travail de Bachelor touche des domaines que j'apprécie énormément (développement *Web* et *3D*). L'implémentation du projet *Virtual-Vertigo* a été agréable et importante car à chaque test de simulation, tout le monde voulait tester également. Il intéresse énormément de gens par son innovation. En effet, il vend du rêve : l'immersion totale dans un monde virtuel. Mon but était d'avoir une simulation opérationnelle même si elle contient encore quelques problèmes. J'ai hâte de le montrer au plus de monde possible et de travailler dans les domaines abordés dans ce travail. Ce fut une expérience très enrichissante et instructive.

J'espère que ce rapport et ce projet sera aussi intéressant pour vous qu'il a été pour moi de le mettre en place et de l'écrire.

Je vous souhaite une bonne lecture.

Informations

António Domingos Ana Sofia : ana.domingos@hotmail.fr

Etudiante ITI travaillant sur le projet *Virtual-Vertigo*

Albuquerque Paul : paul.albuquerque@hesge.ch

Professeur à hebia dans la filière ITI responsable

Gobet Jérémie : jeremy.gobet@hesge.ch

Assistant à hebia dans la filière ITI ayant proposé le sujet et suivant également ce travail

Suivi

Afin de permettre aux personnes suivant le projet de suivre l'avancement du *Virtual-Vertigo* à tout moment, un dépôt *GitHub* a été mis en place. *GitHub* est un service Web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion Git. Il propose un système de *versionning* (cela peut éviter les catastrophes) et un flux donnant la possibilité de partager et suivre des projets.

A noter que le dépôt *GitHub* a été créé lors du travail de semestre et un serveur d'hébergement a été mis en place. Ce serveur a été mis en place parce qu'il était impossible de tester la réalité virtuelle localement. Un système de *WebHook* a été ajouté à *GitHub* afin de pouvoir mettre à jour le serveur hébergeant lors des changements sur le dépôt GitHub. *WebHook* est une méthode permettant d'étendre, de personnaliser et d'intégrer une application Web lorsqu'un événement se produit (pour plus d'informations sur les *WebHook*, voir [1]).

A noter également que le *WebHook* n'est plus utilisé durant le travail de Bachelor. En effet, un serveur *Web* a pris la relais (pour plus d'informations sur ce serveur, voir la section 3.4).

Remerciements

Je tiens à remercier toutes les personnes qui m'ont aidé à la réalisation de ce travail de Bachelor. Plus particulièrement :

M. PAUL ALBUQUERQUE, M. JÉRÉMY GOBET et M. MICHAËL POLLÀ, professeur et assistants à hepia à la filière ITI, pour m'avoir conseillé et aidé tout au long de la réalisation du projet *Virtual-Vertigo*.

M. OLIVIER DONZÉ et M. BENJAMIN DUPOND-ROY, professeur et assistant à hepia à la filière Architecture du paysage, pour m'avoir aidé et modélisé une scène de réalité virtuelle pour le projet *Virtual-Vertigo*.

M. PASCAL REGAMEY, assistant technique à hepia au département construction et environnement, pour avoir fournis la planche de 2.5 mètres pour la simulation de *Virtual-Vertigo*.

Mes collègues de diplômes, pour m'avoir aidé à résoudre des problèmes, pour effectuer des simulations, pour m'avoir consacré du temps et pour la bonne humeur au cours de ce travail de diplôme.

Table des matières

1	Introduction	10
1.1	Contexte	10
1.2	Projet Virutal-Vertigo	11
1.3	Organisation du projet	12
2	Virtual-Vertigo	13
2.1	Dispositif	13
2.2	Architecture	18
3	Composants, technologies et outils	22
3.1	Google CardBoard	22
3.2	Kinect version 1	26
3.2.1	Composants et caractéristiques	26
3.2.2	Capture des données	27
3.2.3	Articulations du squelette et leurs orientations	29
3.2.4	Outils utilisés dans le projet	32
3.3	Technologies 3D	35
3.3.1	X3DOM	35
3.3.2	Logiciels 3D - Blender, MakeHuman et Cinema4D	37
3.4	Serveur Web Virtual-Vertigo en NodeJS	38
3.4.1	NodeJS	39
3.4.2	Module ExpressJS	40
3.4.3	Module Socket.io	40
4	Implémentation	42
4.1	Serveur Virtual-Vertigo	42
4.2	Données provenant du Kinect	44
4.2.1	Capture des données	44
4.2.2	Format des données	45
4.2.3	Traitement des données	46
4.3	Modélisation de réalité virtuelle	47
4.3.1	Création de la scène virtuelle	47
4.3.2	Création des immeubles	48
4.3.3	Création du personnage	49
4.3.4	Textures	50
4.4	Animations sur la réalité virtuelle	51
5	Problèmes rencontrés	58

TABLE DES MATIÈRES

6 Conclusion	64
6.1 Bilan	64
6.2 Améliorations	67
6.3 Perspectives	67
Annexes	68
A Glossaire	68
B Cahier de bord	70
C Planning	75
D Code source	77
E Illustrations différence entre X3DOM et X3D	85
F Fichiers X3DOM	88
G Tutoriel d'installation des outils et utilisation du projet Virtual-Vertigo	92
Bibliographie	98

Table des figures

1.1	Illustration de l'installation mise en place	11
2.1	Vue 3D vue par la personne portant les lunettes	13
2.2	Schéma du projet et de ses composants	14
2.3	Modèle 3D de l'installation de <i>Virtual-Vertigo</i>	15
2.4	Illustrations de l'installation réelle mise en place	16
2.5	Architecture du projet <i>Virtual-Vertigo</i>	18
2.6	Diagramme de séquence du projet <i>Virtual-Vertigo</i>	20
2.7	Diagramme d'activité du projet <i>Virtual-Vertigo</i>	21
3.1	Illustrations des <i>Google CardBoard version 1</i>	22
3.2	Illustrations des Google CardBoard version 2	23
3.3	Stéréoscopie sur smartphone pour <i>Google CardBoard</i>	24
3.4	Illustration de l'application ClassroomVR	25
3.5	Structure du <i>Kinect version 1</i>	26
3.6	Kinect v1 vs Kinect v2	27
3.7	Angles de vue du <i>Kinect</i> horizontal et vertical	28
3.8	Illustration des axes par rapport au <i>Kinect</i>	29
3.9	Articulations du squelette captés par le Kinect	29
3.10	Illustration de la rotation hiérachique	31
3.11	Illustration de l'orientation absolue	31
3.12	Illustration des matrices de rotations en x, y et z	31
3.13	Diagramme de séquences illustrant la communication avec le serveur	38
4.1	Illustration des échanges de données entre serveur, clients HTML et Kinect	43
4.2	Illustration de l'objet json des positions	45
4.3	Illustration de l'objet json de l'orientation des mains	45
4.4	Illustration de l'objet json créé avec les positions reçues et leurs articulations	46
4.5	Modèle de vue 3D modélisé sur Blender	48
4.6	vue entre immeubles et du dessus	49
4.7	vue depuis la rue et depuis la planche	49
4.8	"Evolution" du personnage 3D	49
4.9	Illustrations vue virtuelle avec textures	50
4.10	Illustrations des positions possibles de la caméra	51
4.11	Illustration des axes de l'accéléromètre	52
4.12	Triangle rectangle utilisé pour le calcul de l'angle a	54
4.13	Illustration de l'objet json des articulations reliées	55
4.14	Illustration du cercle trigonométrique des valeurs de atan2	56

TABLE DES FIGURES

5.1	Illustration du triangle rectangle	60
5.2	Rectangle créé sur le sol afin d'effectuer les mesures	61
5.3	Évolution de la distance entre les épaules et la colonne	62
6.1	Illustration des scènes virtuelles 3D créées	64
6.2	Illustration des articulations du squelette capturées par le <i>Kinect v1</i>	65
6.3	Illustration du personnage réaliste créé	65
6.4	Illustration de la vue stéréoscopique sur le smartphone	65
6.5	Illustration du déplacement du squelette 3D dans la réalité virtuelle	66
6.6	Simulation de Virtual-Vertigo	66

Chapitre 1

Introduction

1.1 Contexte

La **réalité virtuelle** est une simulation informatique interactive immersive, en temps réel. Elle peut être constituée d'une simulation visuelle, sonore ou haptique d'environnements réels ou imaginaires. Le but de la réalité virtuelle est de permettre à une personne d'effectuer une activité sensori-motrice et cognitive dans un monde créé numériquement. L'immersion est réalisée à l'aide de dispositifs tels que des casques (*OcculusRift* par exemple), des gants connectés, etc. La réalité virtuelle peut être utilisée pour les **Serious Games** (voir [2]). Le **Serious Game** est un logiciel combinant une intention "sérieuse" avec les jeux par exemple dans un contexte pédagogique ou de rééducation. Il a pour but de rendre attrayants les jeux ludiques. Par exemple, les **Serious Games** peuvent être utilisés pour la santé (le site LudoMedic [3] du studio CCCP [4]).

Le projet *Virtual-Vertigo* est un **Serious Game** mis en place afin d'aider les personnes à combattre le **vertige** en utilisant la réalité virtuelle.

Ce trouble affecte une personne dans le contrôle de sa situation dans l'espace. Il peut survenir dans différentes circonstances ou causes à tout âge. Plusieurs symptômes définissent le vertige :

- une illusion de mouvements,
- une impression de désorientation et déséquilibre,
- une crise de panique,
- bien d'autres.

A noter que les symptômes sont différents pour chaque personne.

Le vertige peut entraîner une phobie : l'acrophobie (la peur du vide). Cette phobie est une peur extrême et irrationnelle des hauteurs. Les acrophobes souffrent de peur panique lorsqu'ils sont en hauteurs et veulent à tout prix redescendre.

Selon les spécialistes, le vertige et l'acrophobie se soignent généralement avec une psychothérapie. Cependant, depuis quelques années, ils commencent à faire des thérapies utilisant la réalité virtuelle. Cette thérapie permet aux personnes d'affronter leurs peurs. A noter que cette thérapie n'a pas encore fait ses preuves. En effet les résultats sont partiels et incertains (pour plus d'informations voir [5]).

1.2 Projet Virutal-Vertigo

Virtual-Vertigo est un projet de développement *Web mobile 3D*. Le but de ce projet est de réaliser un exercice permettant de combattre la peur du vide. En utilisant des casques de réalité virtuelle en carton contenant un smartphone et des lentilles, l'exercice se déroule dans une réalité virtuelle modélisant la vue d'une planche reliant deux immeubles. La personne effectuant la simulation doit traverser cette planche afin d'atteindre l'immeuble d'en face.

Dans ce projet, on emploie un dispositif de *Google CardBoard* pour réaliser le casque de réalité virtuelle. Le monde virtuel 3D sera affiché via la technologie *Web X3DOM* qui permet d'afficher des scènes 3D dans un navigateur. Un utilisateur s'immerge à travers ce dispositif dans le monde virtuel en se déplaçant sur une planche reliant deux immeubles. Un *Kinect* sera utilisé pour récupérer les mouvements et les déplacements de la personne afin de les reporter dans le monde virtuel. Le *Kinect* doit être connecté à un serveur pour transmettre les données capturées aux navigateurs sur smartphone et PC. En outre, des solutions pour rendre l'immersion de l'utilisateur dans le monde virtuel plus réaliste seront abordées.



FIGURE 1.1: Illustration de l'installation mise en place

La figure 1.1 illustre l'installation mise en place pour le projet *Virtual-Vertigo*.

A noter qu'un projet *Ctrl Stress* développé par M. DANIEL MESTRE, psychologue et responsable du Centre de Réalité Virtuelle de Méditerranée, visant à vaincre la peur du vide avec la réalité virtuelle a déjà été mis en place a priori avec des *Occulus Rift*. Pour lire un article sur ce projet, voir [6]. Une étude [7] a également été réalisée sur ce projet.

La principale différence entre *Virtual-Vertigo* et *Ctrl Stress* est que *Virtual-Vertigo* permet à

n’importe quelle personne possédant un *Kinect* et des *Google CardBoard* de l’utiliser alors que *Ctrl Stress* n’est pas disponible. De plus le dispositif de *Crtl Stress* est probablement cher.

A noter également qu’un glossaire décrivant toutes les technologies et les outils utilisés dans le projet *Virtual-Vertigo* est disponible en annexe A.

1.3 Organisation du projet

Le projet *Virtual-Vertigo* dans son intégralité a été séparé en deux parties :

1. Le projet de semestre était d’une durée équivalente à 3 semaines à temps complet, réparties sur un semestre. A la suite de ce travail, les bases du projet ont été mises en place :
 - la construction des *Google CardBoard*,
 - la modélisation d’une vue basique 3D,
 - l’adaptation de la vue aux lunettes (stéréoscopie),
 - le choix entre les différentes technologies possibles pour l’échange et la capture des données,
 - la rotation d’un objet dans la vue 3D en fonction du smartphone,
 - le choix du *Kinect* utilisé,
 - une rapide prise en main du *Kinect*.
2. Lors du travail de Bachelor d’une durée de 12 semaines à temps complet, les éléments suivants ont été réalisés selon des plannings en annexe C :
 - la modélisation de la vue finale (section 4.3.2 et 4.9),
 - le partage des données entre les outils (*Kinect*, serveur et clients HTML) (section 3.4 et 4.1),
 - la création du personnage virtuel (section 4.3.3),
 - le déplacement de la personne dans la réalité virtuelle en fonction de la position de la tête (section 4.4),
 - l’animation des membres de la personne dans la réalité virtuelle (section 4.4),
 - l’adaptation de la vue sur le smartphone en fonction de la rotation de la tête (section 4.4),
 - l’ajout de solutions pour une immersion plus réaliste (section 2.1).

Chapitre 2

Virtual-Vertigo

Ce chapitre décrit de manière plus détaillée le projet *Virtual-Vertigo*. Il est séparé en deux parties distinctes, un descriptif du dispositif et la présentation de l'architecture de l'application. Ces parties sont illustrées afin d'en faciliter la compréhension.

2.1 Dispositif

Le projet *Virtual-Vertigo* aide une personne à combattre sa peur du vide en utilisant la réalité virtuelle. Pour cela, cette personne portera un casque de réalité virtuelle constitué d'un téléphone et d'un support en carton comprenant des lentilles. Ce dispositif permet l'affichage d'une scène virtuelle 3D via des images stéréoscopiques.

Lors de la simulation, l'utilisateur devra avancer sur une planche placée au sol. Un appareil posé sur une table face à la planche permettra de capturer ses mouvements. L'appareil connecté à un serveur devra transmettre les informations de positions au téléphone. Un affichage sur un écran standard donnera aussi une autre vue de la scène.

Voici quelques images illustrant la vue de la scène :

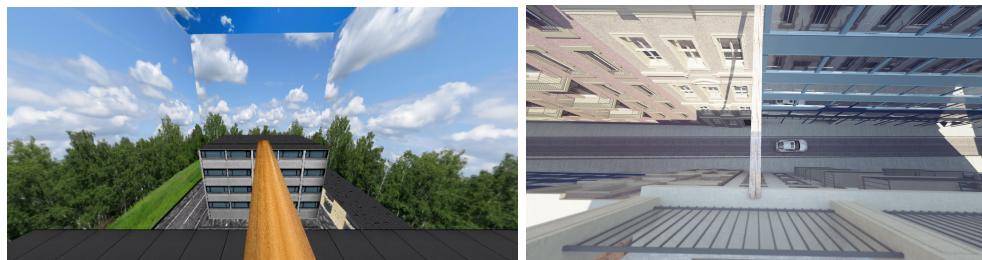


FIGURE 2.1: Vue 3D vue par la personne portant les lunettes

La deuxième image a été produite par M. BENJAMIN DUPOND-ROY, assistant à hepia en filière architecture du paysage. Plus d'informations sont disponibles quant à la construction de ces images à la section 4.3.

A noter qu'un tutoriel contenant les instructions pour l'installation et l'utilisation du projet *Virtual-Vertigo* est disponible en annexe.

Composants

Ci-dessous se trouve la liste des appareils et logiciels utilisés dans le projet :

- **une Google CardBoard** : il s'agit du casque de réalité virtuelle en carton et équipé de lentilles ;
- **un smartphone** : il s'agit de l'appareil qui permet la visualisation en 3D stéréoscopique ;
- **un Kinect version 1** : c'est l'appareil qui détecte les mouvements de la personne sur la planche ;
- **Kinect SDK 1.8** : c'est le logiciel permettant d'utiliser le *Kinect* à partir d'un ordinateur ;
- **NodeJS** : c'est le *Framework JavaScript* permettant de créer et lancer le serveur qui gère les transmissions entre le *Kinect* et les clients HTML (smartphone, PC, etc.) ;
- **un ordinateur sur Windows 7 ou plus** : il s'agit du serveur sur lequel est connecté le *Kinect*.

Le schéma suivant illustre l'interaction entre les composants mentionnés ci-dessus. Sous chaque composant se trouvent les technologies qu'il utilise.

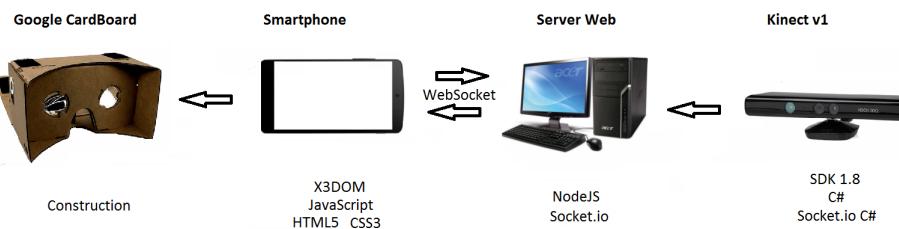


FIGURE 2.2: Schéma du projet et de ses composants

La description de chaque composant est donnée au chapitre 4.
A noter également qu'un glossaire décrivant toutes les technologies et les outils utilisés dans le projet *Virtual-Vertigo* est disponible en annexe.

Prix des composants

Les seuls composants devant être achetés sont le *Kinect* et les *Google CardBoard*, le *smartphone* n'est pas pris en compte dans la liste des achats car tout le monde en possède un. Voici un tableau des prix :

Composant	Prix
Kinect v1 pour Windows	150 CHF
Google CardBoard à assembler	11 CHF
Matériel pour les Google CardBoard	11 CHF
Kinect v2 + adaptateur pour Windows	180 CHF
Occulus Rift	entre 300 et 400 CHF

TABLE 2.1: Tableau prix des composants

Le tableau 2.1 montre que le prix total des composants nécessaires au projet *Virtual-Vertigo* est de moins de 200 CHF. Si l'*Occulus Rift* avait été utilisé au lieu des *Google CardBoard* le prix du projet aurait triplé. Le prix des *Occulus Rift* est économisé.

Disposition générale des composants

Le projet utilise beaucoup de composants qui doivent être positionnés de manière précise. Le schéma suivant montre l'emplacement de chaque composant.

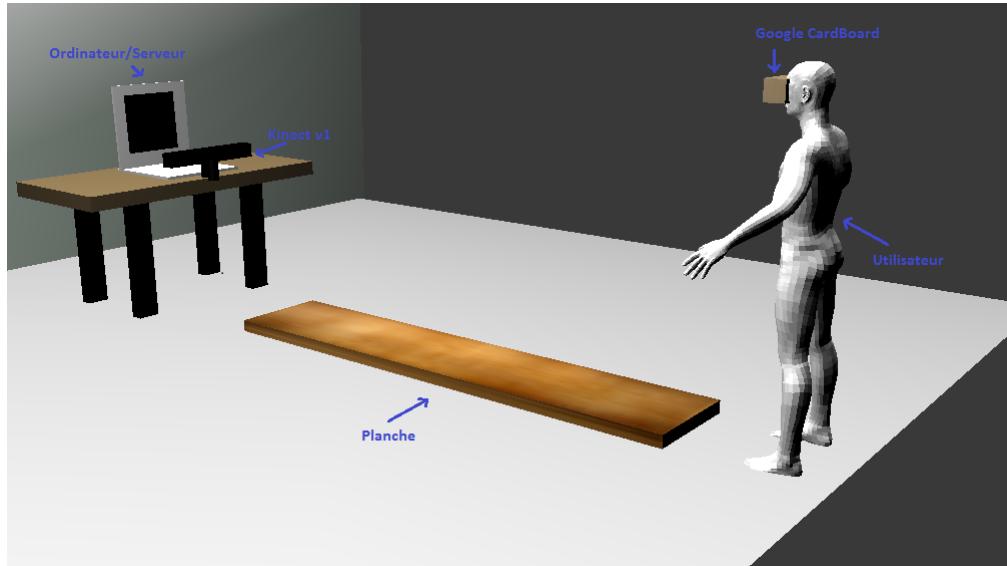


FIGURE 2.3: Modèle 3D de l'installation de *Virtual-Vertigo*

Ce schéma ne montre cependant pas les restrictions imposées par le *Kinect* :

- la planche placée sur le sol ne peut pas faire plus de 3 mètres ;
- la planche doit être placée à une distance d'au moins 80 centimètres du *Kinect* ;
- l'utilisateur doit se trouver à une distance de moins de 4 mètres du *Kinect* ;
- le *Kinect* doit être placé au bord de la table afin d'éviter qu'il ne capte pas les membres inférieurs de l'utilisateur ;
- le *Kinect* doit être connecté en USB à l'ordinateur.

A noter que le smartphone est connecté en wifi ou 3G au serveur. Il est intégré aux *Google CardBoard* et affiche chacune des deux images stéréoscopiques que l'utilisateur visualise à travers les lentilles.

Les photos 2.4 sont l'installation réelle mise en place durant le développement du projet *Virtual-Vertigo*.



FIGURE 2.4: Illustrations de l'installation réelle mise en place

Effets de réalisme

Le réalisme est très important dans ce projet. C'est pour cela que l'utilisation de quelques objets est mis en place afin de plonger la personne dans la situation la plus réaliste possible. Une planche et un effet sonore ont été ainsi ajoutés au projet.

Planche

Pour le ressenti de la planche sous les pieds, une planche de 2,50 mètres de distance ayant environ 5 centimètres d'épaisseur a été mise en place. Cette planche va permettre à la personne de sentir les bords de la planche quand elle s'en approche et donc d'avoir la sensation de déséquilibre et de "chute" possible.

Effet sonore environnant

Lorsque l'utilisateur se trouve sur le toit d'un immeuble, il y a également tous les bruits environnants quotidiens, comme par exemple, les klaxons, les voitures qui passent, les gens qui parlent ou même les oiseaux. Un enregistrement des bruits sonores ambiant peut être effectué, lequel se déclencherait lorsque la personne atteint le bout de la planche. Pour cela, on envisage deux solutions possibles :

- un casque bluetooth qui sera connecté à l'ordinateur ;
- le smartphone lui-même.

Cette amélioration a été mise en place en utilisant le casque bluetooth car sur le smartphone il est impossible de déclencher la bande sonore sans avoir un touch event.

Ce dernier cas sur smartphone n'a pas été envisagé pour une question de temps. Sinon, la bande

peut être lancée lorsque la vue est mise en **fullscreen** manuellement. Le volume serait désactivé tant que la personne n'atteint pas le bout de la planche.

Sensation de vent

Sur le toit d'un immeuble, il y a souvent du vent. C'est pour cette raison qu'un ventilateur peut être ajouté afin que l'utilisateur ressente le vent lors de l'exercice.

Pulsations cardiaques

Afin de pouvoir comparer les résultats des différentes tentatives et donc savoir si le système permet bien de vaincre la peur du vide, une montre connectée peut être utilisée. L'utilisateur devrait alors la porter lors de l'exercice. Cette montre communiquerait au serveur via le **smartphone** les informations cardiaques. Ces informations seraient stockées dans une base de données ou dans un fichier pour le suivi de l'évolution.

Elle permettrait également de contrôler en temps réel la personne en cas de crise ou tout autre problème physique.

2.2 Architecture

L'architecture du projet est importante car elle permet d'avoir une application structurée et complète. L'architecture d'un projet consiste à décrire utilisant des schémas, des graphiques et des diagrammes le projet. Ici, elle contient l'organisation, la description du déroulement du projet et la description de la communication entre chaque composant présent.

Ici, l'organisation consiste à structurer en plusieurs classes ou fichiers. La séparation permet d'avoir une application mieux organisée et modulaire. Cette organisation est décrite dans la figure suivante :

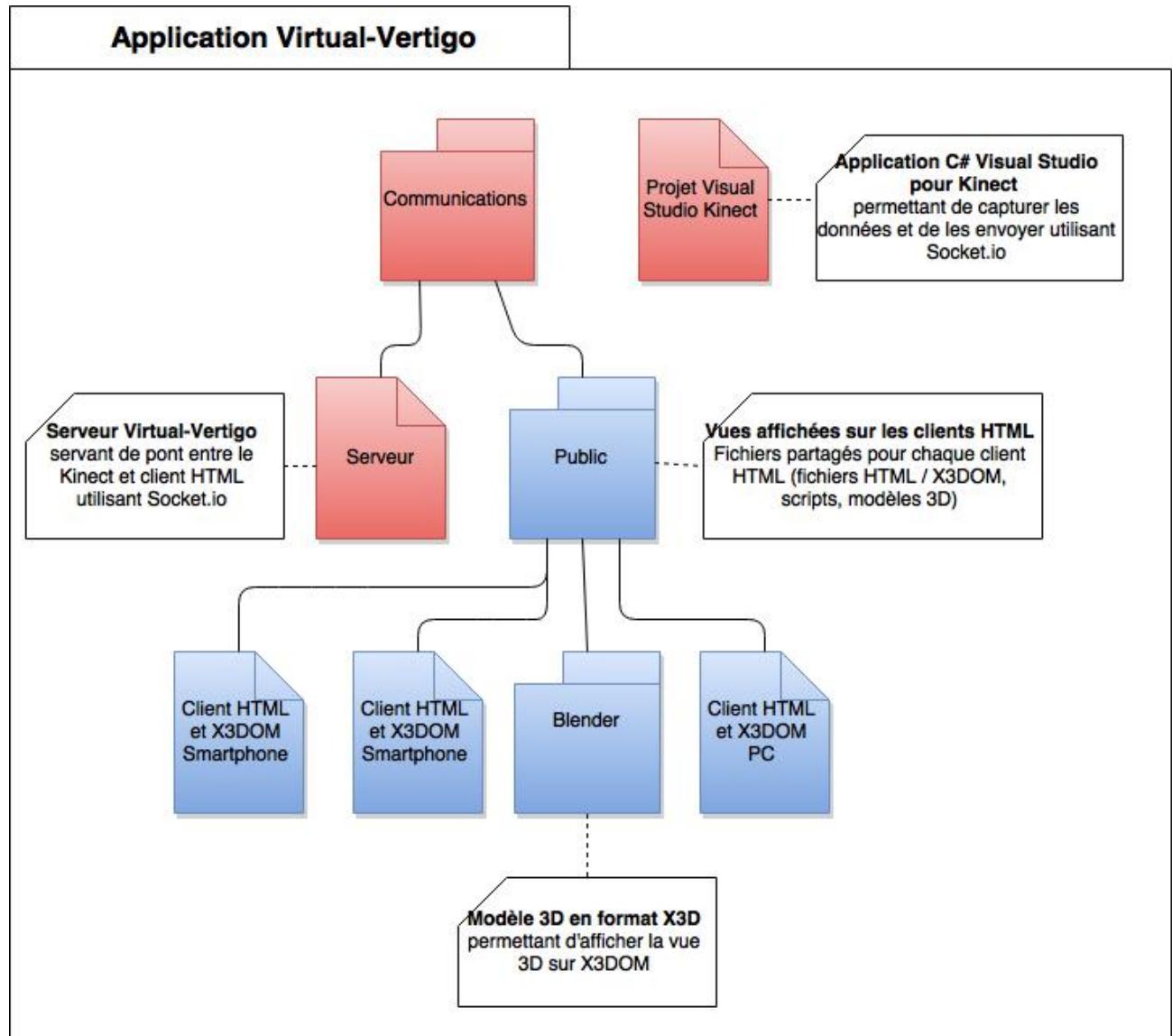
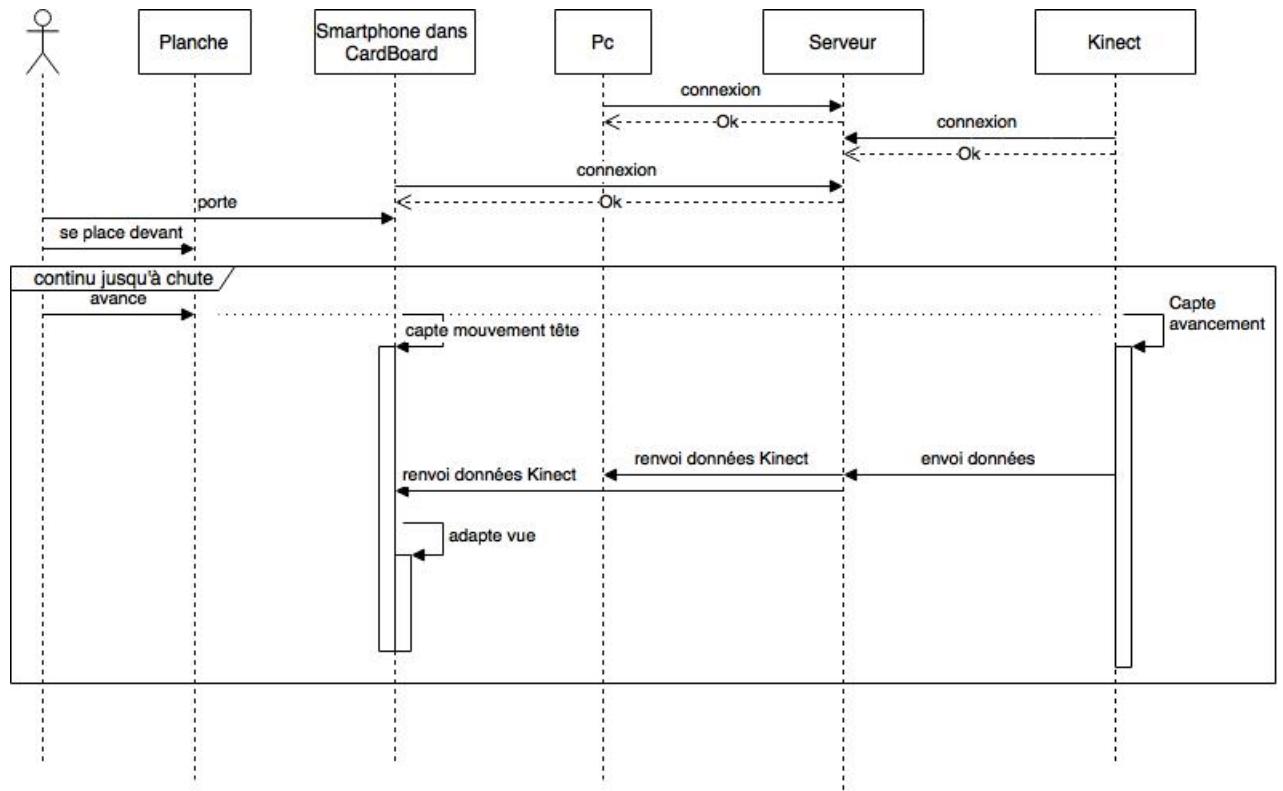


FIGURE 2.5: Architecture du projet *Virtual-Vertigo*

- **Projet Visual Studio Kinect** : il s'agit de l'application C# Visual Studio du *Kinect*. Cette application capte les mouvements de l'utilisateur et envoie les positions aux clients via le module de communication *Socket.io* ;
- **Communications** : il s'agit d'un dossier contenant les fichiers privés de l'application qui gèrent la communication. Ce dossier contient aussi les fichiers nécessaires au fonctionnement du serveur ;
- **Serveur** : il s'agit du *serveur Web Virtual-Vertigo*. Ce serveur correspond au pont entre le *Kinect* et les clients HTML ;
- **Public** : c'est le dossier contenant les fichiers publics de l'application. Ce dossier contient les pages HTML, les scripts et les modèles 3D ;
- **Script** : il s'agit du script modifiant la scène 3D. Ce script réceptionne les positions, crée le personnage virtuel, anime les membres de ce personnage et déplace ce personnage sur la planche ;
- **Client HTML et X3DOM Smartphone** : c'est le fichier HTML contenant la scène stéréoscopique 3D ;
- **Blender** : c'est le dossier contenant les modèles 3D ainsi que leurs textures ;
- **Client HTML et X3DOM PC** : c'est le fichier HTML contenant la scène non stéréoscopique 3D affichée sur un écran standard.

A noter que les éléments en rouge sont des éléments privés que seul la personne s'occupant du serveur peut avoir accès. Les éléments en bleu contiennent les éléments publiques. C'est-à-dire, les clients HTML (*smartphone* et *PC*), les scripts JavaScript interagissant avec la vue et les modèles 3D. Plus de détails sur ces composants logiciels sont donnés au chapitre 4.

La description du déroulement du projet consiste, via un diagramme ou un schéma, à montrer la vie du projet. C'est à dire, décrire ce qu'il se passe entre chaque événement et comment le changement d'événement s'effectue. Le déroulement du projet est illustré par le diagramme de séquence 2.6 :

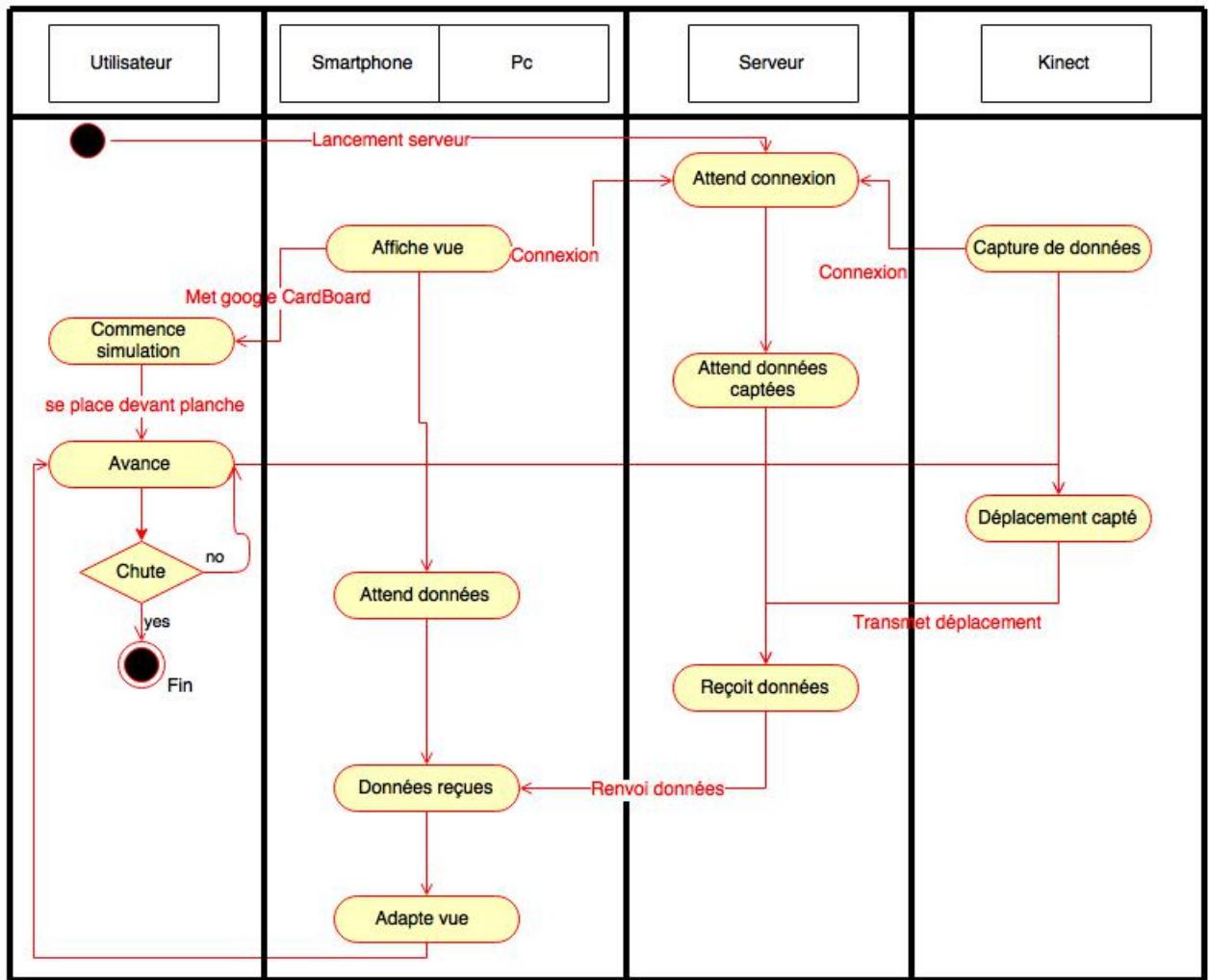

 FIGURE 2.6: Diagramme de séquence du projet *Virtual-Vertigo*

Ce diagramme montre deux phases : l'une d'initialisation, et l'autre de simulation. L'initialisation correspond au lancement du serveur, puis la connexion du smartphone et du *Kinect* à ce serveur. La phase de simulation commence quand l'utilisateur se place devant la planche. Une fois la simulation démarrée, les opérations suivantes sont effectuées en boucle :

- le *Kinect* capte les mouvements de la personne ;
- le *Kinect* transmet les positions au serveur ;
- le serveur retransmet ces positions aux clients HTML connectés ;
- Les clients HTML adaptent leurs vues.

Cette boucle s'exécute toutes les secondes et s'arrête quand la personne "chute" ou quand elle arrive au bout de la planche.

Le déroulement du projet peut également être détaillé en se basant sur les différents états durant la simulation. Le diagramme 2.7 montre les états des composants du projet et l'action qui les fait changer d'état.


 FIGURE 2.7: Diagramme d'activité du projet *Virtual-Vertigo*

L’application commence par le lancement du serveur qui se met en attente d’une connexion d’un client. Lorsque le *Kinect* et les clients HTML se connectent, le serveur attend de recevoir les positions captées par le *Kinect* pour ensuite les retransmettre aux clients HTML. Lors de la réception des positions, les clients adaptent la vue virtuelle. En parallèle, l’utilisateur effectue sa simulation de réalité virtuelle.

Chapitre 3

Composants, technologies et outils

Ce chapitre décrit tous les composants, technologies et outils nécessaires à la réalisation du projet *Virtual-Vertigo*. Pour chaque section, un exemple d'utilisation général a été inclus.

3.1 Google CardBoard

Les *Google CardBoard* sont des casques de réalité virtuelle en carton équipé de deux lentilles et dans lesquels peut venir se loger un smartphone. Elles se construisent à la maison avec très peu de matériel et sont accessibles à tout le monde à un prix très bas. Google permet ainsi via ses lunettes de pouvoir utiliser des applications de réalité virtuelle 3D sans avoir à mettre une somme d'argent trop importante.

Version 1



FIGURE 3.1: Illustrations des *Google CardBoard version 1*

La première version des *Google CardBoard* a été introduite en 2014.

Matériel pour le montage

Pour la construction, il ne faut pas grand chose :

- du carton pas trop épais (1-2 millimètres)
- deux lentilles biconvexes
- un tag NFC
- un aimant céramique
- un aimant néodyme
- du velcro

A noter que les aimants servent à interagir avec le smartphone pour actionner les boutons.

Les patrons de conception et les instructions de montage sont mis à disposition sur le site des *Google CardBoard* [8]. Il suffit alors de les imprimer, les coller sur le carton, découper et assembler tout les composants pour obtenir des *Google CardBoard*. Il est également possible d'acheter des *Google CardBoard* pré-fabriquées.

Dans le cadre du Projet *Virtual-Vertigo*, les *Google CardBoard* utilisées ont complètement été construites manuellement. Cependant, seules les lentilles biconvexes ont servi. En effet, l'aimant et le patch NFC n'ont pas été utilisé pour *Virtual-Vertigo*.

A noter qu'un tutoriel contenant les instructions pour le montage des *Google CardBoard* est disponible en annexe.

Version 2

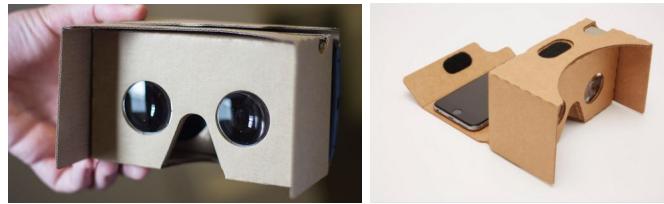


FIGURE 3.2: Illustrations des Google CardBoard version 2

La nouvelle version des Google CardBoard est sortie en juin 2015

Nouveautés

- Cette version est capable de contenir un smartphone jusqu'à 6 pouces.
- Les aimants étant pas fiable sur la version 1 ont été remplacés par un bouton physique interagissant directement avec le smartphone.
- Google a implémenté un SDK qui supporte les iPhones et les Android.
- Le montage est plus simple.

Matériel pour le montage

Google n'a pas encore transmis les patrons de conception ni la liste de matériel pour construire cette nouvelle version.

Vision stéréoscopique sur le smartphone

La stéréoscopie est une technique mise en oeuvre pour reproduire une perception de 3D à partir de deux images. Elle se base sur le fait que la perception humaine de la 3D se forme dans le cerveau lorsqu'il reconstitue une seule image à partir de la perception de deux images provenant de chaque oeil.

La stéréoscopie pour les *Google CardBoard* s'effectue sur le smartphone. La vue affichée sur le smartphone doit avoir une vue pour chaque oeil. Puis c'est par l'intermédiaire des lentilles que ses images planes seront perçues comme des images 3D.

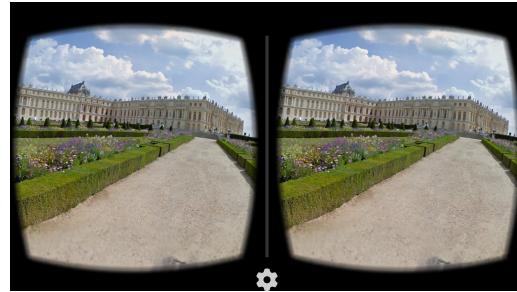


FIGURE 3.3: Stéréoscopie sur smartphone pour *Google CardBoard*

Pour plus d'informations sur la stéréoscopie, voir [9].

Applications sur Google CardBoard

La vue sur le smartphone doit absolument être stéréoscopique afin que les lentilles donnent un effet de 3D. Ainsi, en plaçant le smartphone dans les *Google CardBoard* avec une application pour CardBoard lancée, l'immersion dans le monde virtuel est immédiate.

Une application utilisant les *Google CardBoard* doit être stéréoscopique et peut par exemple être implémentée en deux langages différents :

- comme une application Android en Java : cela implique que tout est stocké et exécuté sur le smartphone ;
- comme une application Web via un Framework 3D : cela implique que le smartphone charge seulement une page Web.

Applications disponibles

Plusieurs applications sont disponible pour les *Google CardBoard* implantée sur Android ou le Web. Ci-dessous, la description d'une application Android suivie de la description d'une application Web pour les *Google CardBoard*.

L'application Android CardBoard disponible sur le play store permet de tester plusieurs démonstrations sur les Google CardBoard. Pour vous procurer l'application CardBoard, voir [10]. Ci-dessous la liste des démonstrations que l'application propose :

- **Earth** : aller où vous voulez avec Google Earth ;
- **Tour Guide** : visiter Versailles avec un guide ;
- **My Videos** : regarder vos vidéos sur un grand écran ;
- **Exhibit** : examiner un artefact culturel sous différents angles ;
- **Photo Sphere** : regarder vos photos sphère ;
- **Windy Day** : suivre une histoire dans une courte scène interactive de Google Spotlight Stories, votre cinéma de poche (pour plus d’informations sur Google Spotlight Stories, voir [11]).

L’application ClassroomVR disponible sur internet utilise le *Framework X3DOM* (voir section 3.3.1). Cette application met en scène une salle de classe stéréoscopique 3D. Ci-dessous une illustration de cette application :



FIGURE 3.4: Illustration de l’application ClassroomVR

La scène 3.4 met en scène une salle de classe contenant des tables, des chaises et un tableau noir. Lorsque cette vue est visualisée via les lentilles des *Google CardBoard*, la personne est plongée dans cette salle et peut l’observer depuis un point fixe.

3.2 Kinect version 1

Le *Kinect* est un périphérique initialement destiné à la console de jeux Xbox 360. Il permet à l'utilisateur d'interagir en utilisant son corps plutôt qu'une manette. Depuis quelques années, Microsoft a mis à disposition un SDK permettant d'utiliser le *Kinect* sur un ordinateur Windows. Cependant, le *Kinect* nécessite des caractéristiques hardware spécifiques (pour plus d'informations sur les caractéristiques du *Kinect v1*, voir [12]).

Le *Kinect* a été choisi pour capter et reporter les mouvements et le déplacement de la personne. Ainsi grâce au *Kinect*, la personne peut voir ses mains ou ses pieds (en simplifié) lorsqu'il avance sur la planche. Le *Kinect version 1* a été choisi en particulier car l'école disposait déjà d'un tel appareil et également parce qu'il convenait amplement aux besoins de *Virtual-Vertigo*.

A noter qu'un tutoriel contenant les instructions pour l'installation et l'utilisation du *Kinect v1* est disponible en annexe.

3.2.1 Composants et caractéristiques

Cette section contient une description succincte des composants et caractéristiques du *Kinect v1*.

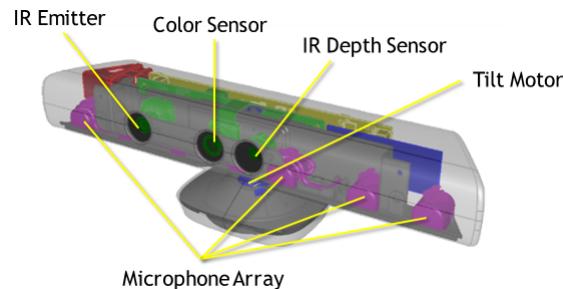


FIGURE 3.5: Structure du *Kinect version 1*

- **Color Sensor** : caméra RGB
- **IR Emitter** : émet des infrarouges pour le Depth Sensor
- **IR Depth Sensor** : lit les infrarouges renvoyés au Kinect, puis les convertit en information de profondeur mesurant la distance entre l'objet et le Kinect
- **Tilt Motor** : permet de changer et de déterminer l'orientation du *Kinect v1* sur un axe 3D
- **Microphone Array** : 4 microphones qui permettent de capturer le son

Max Distance	4.5 m
Min Distance	0.8 m (normal mode) and 0.4 m (near mode)
Horizontal Field Of View	57 degrees
Vertical Field Of View	43 degrees
Frame Rate	30 fps
Skeleton Joints Defined	20
Full Skeleton Tracked	2
Audio Format	16 kHz (24 bit mono PCM)
Vertical Tilt Range	+ - 27 degrees

 TABLE 3.1: Tableau illustrant les caractéristiques du *Kinect v1*

Comparaison v1 et v2



FIGURE 3.6: Kinect v1 vs Kinect v2

- la plus grande différence entre les 2 versions est la résolution.
- La résolution couleur est plus de 3 fois plus élevée et la résolution de profondeur est quasiment 2 fois plus élevée.
- les champs de vue horizontal et vertical ont également augmenté.
- 25 articulations sont captées par le *Kinect v2* contre 20 pour le *Kinect v1*.
- le principal défaut du *Kinect v2* est que les caractéristiques hardware sont plus exigeantes.
- le nombre de fps (Frame par secondes) n'a pas changé entre les 2 versions.

3.2.2 Capture des données

Le *Kinect* permet de capter différents types de données d'une personne de plusieurs manières. Les différentes captures se font en utilisant les capteurs du *Kinect*.

Le *Kinect* permet de capturer les données d'une personne dans 2 positions différentes : **default**, la personne est debout, et **seated**, la personne est assise. Les informations ci-dessous peuvent être capturés dans les deux positions.

Angle de vue

Le *Kinect* capte les informations sur un rayon de 43 degrés à la verticale et de 57 degrés à l'horizontale. L'angle de vue du *Kinect* se traduit comme un cône, le *Kinect* étant la partie la plus étroite du cône.

Ci-dessous une illustration des angles de vue du *Kinect*.

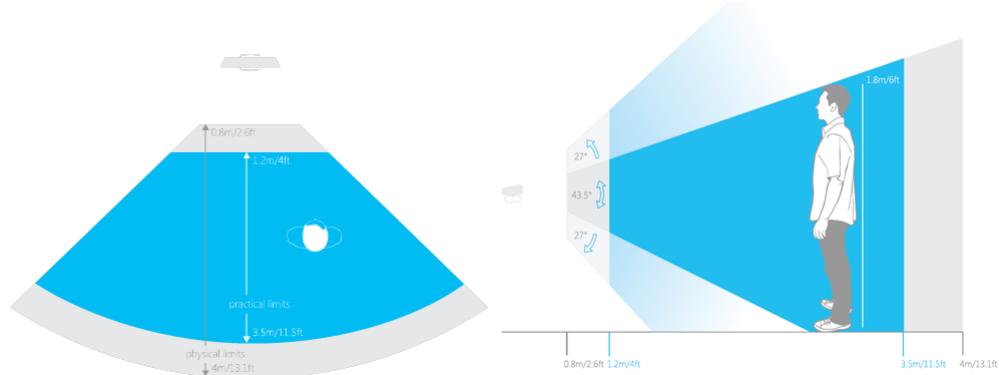


FIGURE 3.7: Angles de vue du *Kinect* horizontal et vertical

Color Stream

Le Color Stream sert de caméra RGB au *Kinect*. Il retransmet les informations que le Color Sensor capte. Il permet donc d'afficher et de modifier la vue en temps réel.

Il permet d'utiliser plusieurs formats différents :

- Raw YUV, résolution de 640×480 à 15 fps
- RGB, résolution de 1280×960 à 12 fps
- RGB, résolution de 640×480 à 30 fps
- YUV, résolution de 640×480 à 15 fps
- Infrared de type MONO16, résolution de 640×480 à 30 fps permet de capter en cas de lumière faible
- RawBayer de type MONO8, résolution de 1280×960 à 12 fps ou de 640×480 à 30 fps pour utiliser son propre algorithme de reconstruction d'image

Le Color Sensor n'est pas utilisé dans ce projet.

Depth Stream

Le Depth Stream sert à capter les informations en utilisant la profondeur détectée par les infrarouges. Il permet donc d'afficher et de modifier la vue en profondeur avec des nuances de gris. Il capte la profondeur en deux modes : `default`, de 0.6 à 4m, et `near`, de 0.4 à 3m.

Il permet également de le faire à 30 fps en différentes dimensions : 640×480 , 320×240 et 80×60 . Le Depth Stream n'est pas utilisé dans ce projet.

Skeletal Stream

Le **Skeletal Stream** retourne les positions du squelette ainsi que la position, en 3 dimensions, de chacune des 20 articulations de référence sur le squelette en mètres (voir section 3.2.3).

Il faut savoir que l'origine est le *Kinect*, que les axes x et y sont les offset du centre de vision du *Kinect* et que l'axe z est la distance entre le *Kinect* et l'objet.

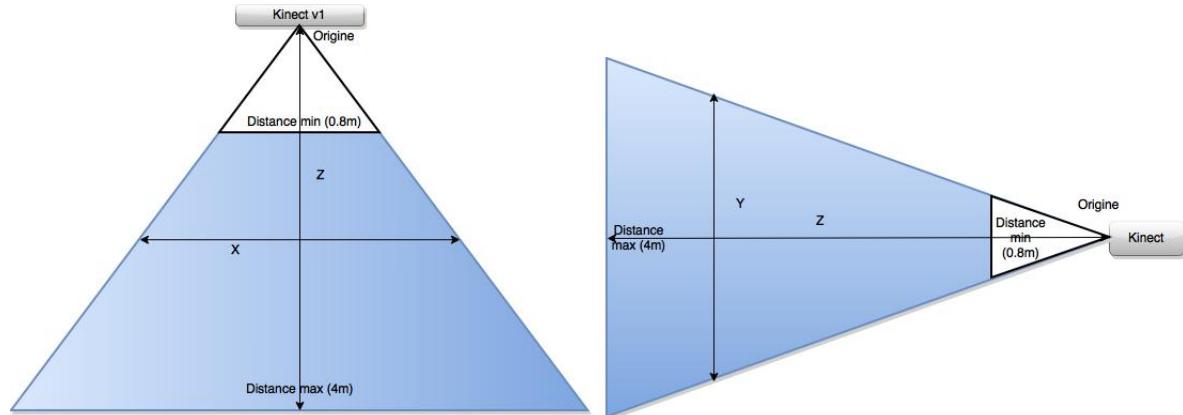


FIGURE 3.8: Illustration des axes par rapport au *Kinect*

Le figure 3.8 illustre comment sont placé les axes x, y et z en fonction du *Kinect*. La partie verte sur chaque figure correspond à la zone que le *Kinect* capte.

3.2.3 Articulations du squelette et leurs orientations

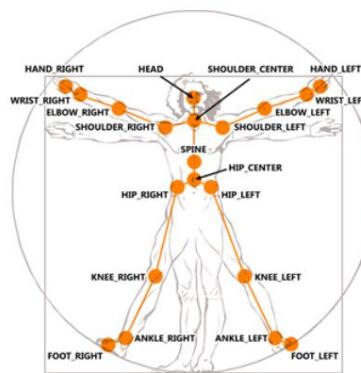


FIGURE 3.9: Articulations du squelette captés par le *Kinect*

Les 20 articulations que le *Kinect v1* détecte se trouvent sur l'illustration 3.9. A chacune de ces articulations correspond un identifiant qui permet ensuite de retrouver le nom de l'articulation détectée. Le tableau 3.2 ci-dessous donne ces valeurs.

Identifiant	Articulation	Nom sur squelette
0	base of the spine	hipcenter
1	middle of the spine	spine
2	neck	shouldercenter
3	head	head
4	left shoulder	shoulderleft
5	left elbow	elbowleft
6	left wrist	wristleft
7	left hand	handleft
8	right shoulder	shoulderleft
9	right elbow	elbowright
10	right wrist	wristright
11	right hand	handright
12	left hip	hipleft
13	left knee	kneeleft
14	left ankle	ankleleft
15	left foot	footleft
16	right hip	hipright
17	right knee	kneeright
18	right ankle	ankleright
19	right foot	footright
20	spine at the shoulder	n'est pas détecté
21	tip of the left hand	n'est pas détecté
22	left thumb	n'est pas détecté
23	tip of the right hand	n'est pas détecté
24	right thumb	n'est pas détecté

TABLE 3.2: Tableau illustrant les valeurs de chaque articulation

Le *Kinect v1* capture également l'orientation des articulations du squelette. Connaître l'orientation des articulations du squelette permet d'effectuer une plus précise animation d'un avatar. Le *Kinect v1* peut capter l'orientation de deux manières : la rotation hiérarchique, basé sur les membres reliés, et l'orientation absolue, utilisant les coordonnées de la caméra.

Les deux images ci-dessous illustrent comment le *Kinect v1* capture l'orientation des articulations du squelette :

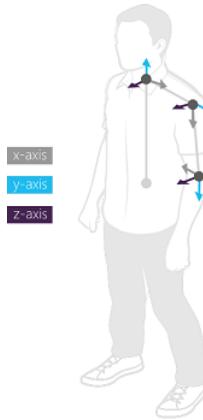


FIGURE 3.10: Illustration de la rotation hiérachique

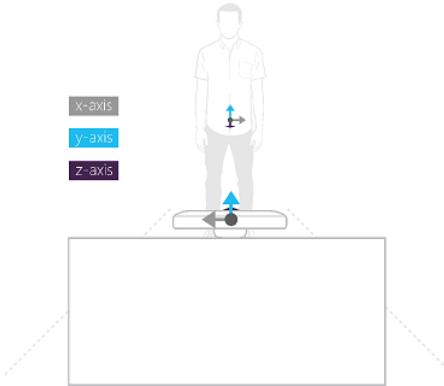


FIGURE 3.11: Illustration de l'orientation absolue

Pour plus d'informations sur les orientations des articulations, voir [13]. Ces données d'orientations peuvent être retournées sous forme de quaternions ou de matrices de rotations. Ci-dessous une illustration des deux formes :

$$q = r + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \quad (3.1)$$

tel que r est un réel consistant la partie réelle et $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ consistant la partie pure d'Hamilton, où : $\mathbf{i} = (1,0,0), \mathbf{j} = (0,1,0), \mathbf{k} = (0,0,1)$.

$$\begin{aligned} R_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \\ R_z(\theta) &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

FIGURE 3.12: Illustration des matrices de rotations en x, y et z

Pour plus d'informations sur les quaternions, voir [14] et pour plus d'informations sur les matrices de rotations, voir [15].

3.2.4 Outils utilisés dans le projet

Framework .NET

Le *Framework .NET* est utilisé par Windows. Il s'appuie sur une norme CLI (Common Language Infrastructure) qui est indépendante du langage de programmation. Cela permet aux langages compatibles respectant cette norme d'avoir accès à toutes les bibliothèques disponibles sur l'environnement d'exécution. Il fournit une approche unifiée pour la conception d'applications Windows et Web.

Visual Studio 2013

Visual Studio 2013 est un logiciel de développement pour Windows conçue par Microsoft. Il permet de générer des applications Web ASP.NET, des services Web XML et bien d'autres. Les langages tel que *Visual Basic*, *Visual C++*, *Visual C#* et *Visual J#* permettent de mieux tirer parti des fonctionnalités du *Framework .NET*.

SDK 1.8 du Kinect v1

Le *SDK 1.8 pour Windows* fournit des outils et une API pour implémenter une application utilisant le *Kinect*.

Le *SDK 1.8* est la mise à jour des anciens SDK. En plus des autres SDK, il contient :

- **Kinect Background Removal** : fournit un écran vert pour une seule personne.
- **Webserver for Kinect Data Streams** : permet l'envoi des informations captées par le *Kinect* sur une page Web utilisant les *WebSockets*.
- **Color Capture and Camera Pose Finder for Kinect Fusion** : permet de scanner un objet 3D puis de le modéliser en utilisant le *Kinect*.
- Beaucoup d'exemples.

Pour plus d'informations sur les différents *SDK du Kinect v1*, Voir [16].

Ce SDK est le dernier que Windows a sorti pour le *Kinect v1*. Quand le *Kinect v2* est sorti le SDK 2.0 a pris le relais.

Exemple d'utilisation

L'exemple qui suit capte la personne devant le Kinect. Il est séparé en deux parties : l'initialisation du *Kinect* et la capture des données.

```
// Initialisation du Kinect
KinectSensor kinect = null;

void StartKinectST()
{
    // Get first Kinect Sensor
    kinect = KinectSensor.KinectSensors.FirstOrDefault(s => s.Status ==
        KinectStatus.Connected);
    kinect.SkeletonStream.Enable(); // Enable skeletal tracking

    // Allocate ST data
    skeletonData = new
        Skeleton[kinect.SkeletonStream.FrameSkeletonArrayLength];

    // Get Ready for Skeleton Ready Events
    kinect.SkeletonFrameReady += new
        EventHandler<SkeletonFrameReadyEventArgs>(kinect_SkeletonFrameReady);

    // Start Kinect sensor
    kinect.Start();
}

//Capture des donnees
private void kinect_SkeletonFrameReady( object sender, SkeletonFrameReadyEventArgs e)
{
    // Open the Skeleton frame
    using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
    {
        // check that a frame is available
        if (skeletonFrame != null && this.skeletonData != null)
        {
            // get the skeletal information in this frame
            skeletonFrame.CopySkeletonDataTo(this.skeletonData);

            // draw the skeletons
            foreach (Skeleton skeleton in this.skeletonData)
            {
                if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
                {
                    DrawSkeletonPosition(skeleton.Position);
                }
            }
        }
    }
}
```

Socket IO C# Client

Socket IO C# Client (socket.io-csharp-client) est un projet disponible sur *GitHub* [17]. Il permet de faire communiquer un programme C# avec un *serveur NodeJS*. Il utilise le package *NuGet*, un gestionnaire de packages sur *Visual Studio*. Il permet d'ajouter des packages à une application *Visual Studio*.

Il permet de créer et modifier des packages et fournit un grand nombre de packages pour Windows.

Il utilise également *WebSocket4Net*, un autre projet *GitHub* [18] qui permet d'utiliser les *Web-Sockets* client.

Ce projet est utilisé pour transmettre les données captées au client HTML. Il a permis de résoudre le problème décrit au chapitre 5

Exemple d'utilisation

L'exemple ci-dessous n'utilise pas le *Kinect*. Il montre comment utiliser *SocketIOC# Client* dans un projet.

```
using System;
using SocketIO.Client;

namespace SimpleClient
{
    class Example
    {
        static void Main()
        {
            var io = new SocketIOClient();

            var socket = io.Connect("http://localhost:3000/");

            socket.On("data", (args, callback) =>
            {
                Console.WriteLine("Server sent:");

                for (int i = 0; i < args.Length; i++)
                {
                    Console.WriteLine("[" + i + "] => " + args[i]);
                }
            });

            string line;

            while ((line = Console.ReadLine()) != "q")
            {
                socket.Emit("data", line);
            }
        }
    }
}
```

3.3 Technologies 3D

Ce projet nécessite la visualisation de scènes 3D et la modélisation de scènes 3D. Le *Framework X3DOM* permet à ce projet de ne pas être dépendant du système d'exploitation du smartphone car il suffit d'un accès internet. Cette solution permet à n'importe qui de pouvoir facilement tester et utiliser ce projet. Il permet également d'avoir plusieurs clients, un testeur et une visionner.

Les logiciels de modélisations 3D *Blender* et *Cinema4D* ont été utilisés pour modéliser des scènes 3D. *Blender* permet d'exporter les scènes 3D en format X3D supporté par *X3DOM*. Le personnage 3D a été modélisé en utilisant le logiciel *MakeHuman*. Plus d'informations sur la modélisation des scènes du projet au chapitre 4.

3.3.1 X3DOM



X3DOM est un Framework open source qui permet d'afficher des scènes 3D sur une page Web. *X3DOM* est la composition entre X3D (Extensible 3D Graphics) et DOM (Document Object Model). Il permet d'intégrer X3D dans le DOM et d'utiliser, par exemple un listener d'événements ou les modifications du HTML dynamiquement avec JavaScript.

L'utilisation de *X3DOM* au lieu d'une autre librairie apporte certains avantages :

- pas besoin d'un plugin pour afficher les scènes 3D ;
- étant basé sur un nouveau profil HTML de l'ISO Standard X3D, il est conforme au standard ;
- il existe une large communauté de développeurs ;
- utilisation du HTML et DOM plutôt que d'apprendre à programmer avec une nouvelle API ;
- il suffit de télécharger et inclure les documents CSS et JavaScript pour l'utiliser.

Pour plus d'informations sur *X3DOM* voir [19], sur X3D voir section 3.3.2 ou sur DOM voir [20].

Exemple d'utilisation

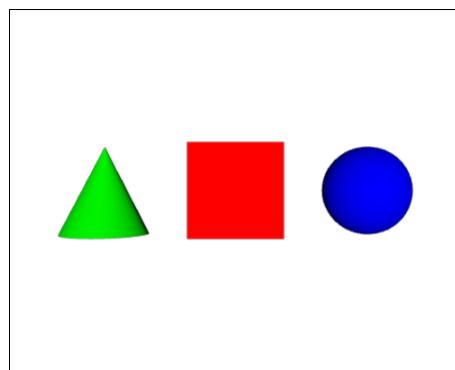
Afin d'ajouter de la 3D sur une simple page HTML, il faut impérativement avoir les éléments suivants :

- le fichier de script x3dom.js permettant de manipuler la 3D,
- le fichier de style x3dom.css pour le placement des éléments dans la scène 3D.

Voici un exemple très simple d'utilisation de *X3DOM*. Dans cet exemple, un cône, un cube et une sphère sont créés sur la même ligne et bougent tous en même temps.

```
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <title>My first X3DOM page</title>
    <script src='http://www.x3dom.org/download/x3dom.js'></script>
    <link rel='stylesheet' type='text/css'
          href='http://www.x3dom.org/download/x3dom.css' />
  </head>
  <body>
    <x3d width='500px' height='400px'>
      <scene>
        <shape>
          <appearance>
            <material diffuseColor='1 0 0'></material>
          </appearance>
          <box></box>
        </shape>
        <transform translation=' -3 0 0'>
          <shape>
            <appearance>
              <material diffuseColor='0 1 0'></material>
            </appearance>
            <cone></cone>
          </shape>
        </transform>
        <transform translation=' 3 0 0'>
          <shape>
            <appearance>
              <material diffuseColor='0 0 1'></material>
            </appearance>
            <sphere></sphere>
          </shape>
        </transform>
      </shape>
    </scene>
  </x3d>
  </body>
</html>
```

Voici le résultatat de cet exemple :



3.3.2 Logiciels 3D - Blender, MakeHuman et Cinema4D

Blender

Blender est un logiciel de modélisation, d'animation et de rendu 3D développé par la Fondation Blender.

X3DOM prévoit une balise utilisant un format que seul *Blender* met à disposition en exportation au format X3D.

Format x3d

Le format *X3D* (Extensible 3D) est un format de fichier graphique et multimédia orienté 3D. X3D s'appuie sur une structure de type graphe de scène.

X3D est un couplage entre une page HTML et un objet *Blender*.

Le format X3D est très similaire à *X3DOM*. Un exemple du format X3D et un exemple de *X3DOM* sont placés en annexes afin de voir la comparaison (pour plus d'informations sur X3D voir [21]).

MakeHuman

MakeHuman est un logiciel libre de modélisation 3D de corps humains. Les modèles générés sont destinés à être utilisés sur *Blender* ou un autre logiciel 3D.

Il dispose d'une interface simple permettant de créer facilement un corps humain de n'importe quel type et en modifiant et personnalisant toutes les caractéristiques du corps humain.

Cinema4D

Cinema4D est un logiciel de création 3D développé par Maxon. Il permet la modélisation, le texturage, l'animation et le rendu d'objets 3D.

3.4 Serveur Web Virtual-Vertigo en NodeJS

Dans le cadre de ce projet, un serveur est nécessaire pour transmettre les informations captées par le *Kinect* aux clients HTML.

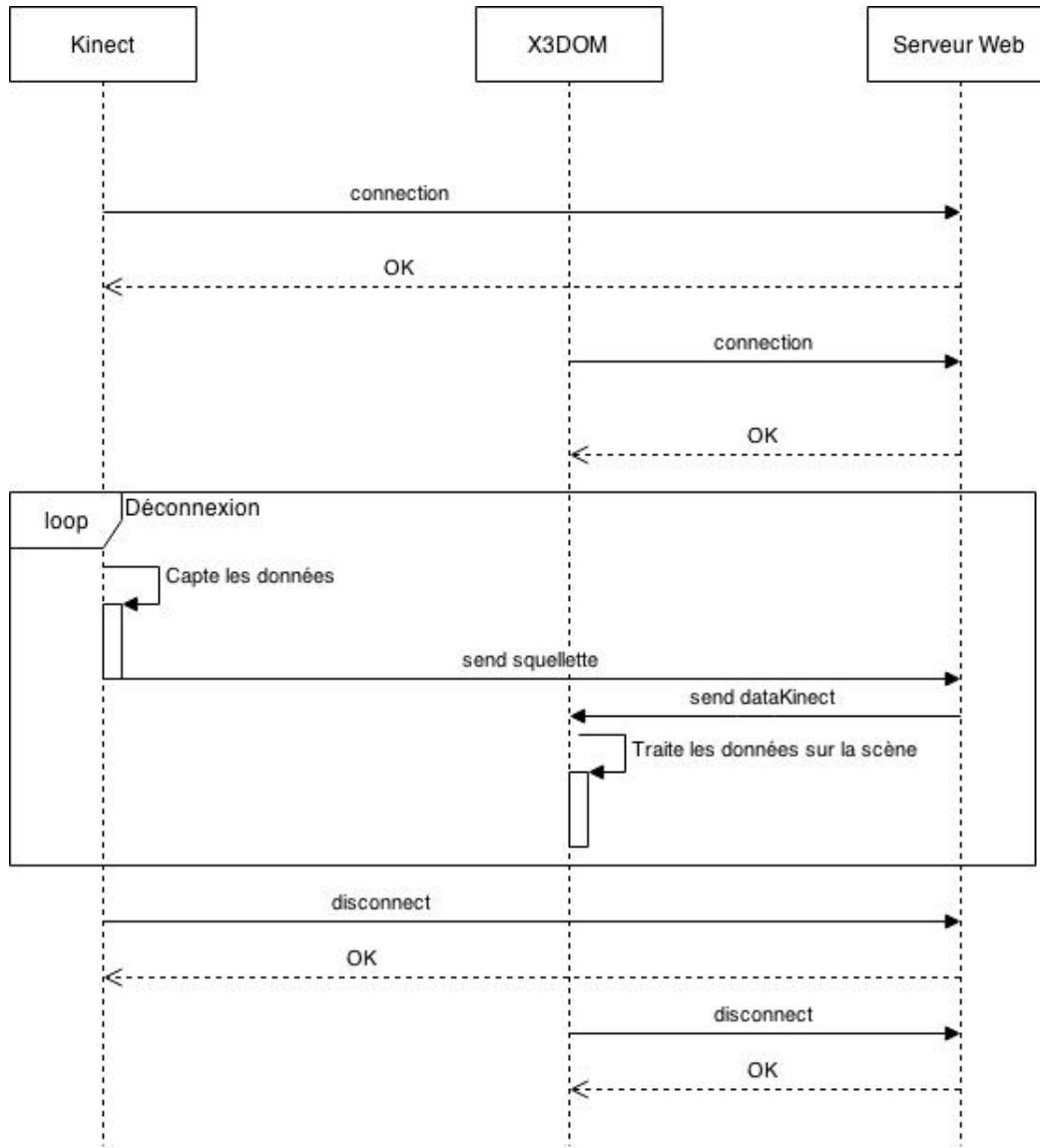


FIGURE 3.13: Diagramme de séquences illustrant la communication avec le serveur

Comme vous pouvez le voir sur ce diagramme de séquences 3.13, le *Kinect* et le client HTML se connectent au serveur. Puis le *Kinect* envoie toutes les secondes les informations captées et dès que le serveur reçoit les données du *Kinect*, il les retransmet directement aux clients HTML qui va les interpréter sur la scène 3D. Tout ceci s'effectue jusqu'à la déconnexion du *Kinect* et des clients HTML.

Le *serveur Web Virtual-Vertigo* est un serveur Web utilisant les *WebSockets* pour les communications. Lors du projet de semestre sur le même thème, un travail préliminaire a permis de choisir entre *MeteorJS* et *NodeJS*. *MeteorJS* est un Framework permettant de construire des sites Web modernes et d'implémenter automatiquement un serveur Web, sans avoir besoin de gérer quoi que ce soit. Cependant, *NodeJS* a été choisi car on avait besoin de plus de contrôle sur l'implémentation du serveur.

Pour plus d'informations sur les *WebSockets* voir [22], sur *MeteorJs* voir [23].

3.4.1 NodeJS



NodeJS est un Framework logiciel et évènementiel en JavaScript. Il contient une bibliothèque de serveur HTTP, ce qui permet de faire tourner et de mieux contrôler un serveur Web sans avoir besoin d'un logiciel externe.

L'avantage de ce Framework est qu'il permet d'ajouter plusieurs modules permettant de manipuler les services Web plus facilement (par exemple l'utilisation de *WebSockets* avec le module *Socket.io*). Ces modules sont disponibles car il utilise *npm*, un gestionnaire de packages officiel pour *NodeJS* qui permet d'installer des applications disponibles sur le dépôt *npm*.

A noter que la plupart des projets disponibles sur le dépôt *npm* sont des projets implémentés par un utilisateur qui a voulu partager son travail en le mettant à disposition sur *GitHub* (pour plus d'informations sur *NodeJS*, voir [24]).

Exemple d'utilisation

NodeJS contient deux modules déjà installés :

- http : module permettant de créer un serveur *http*
- net : module permettant de créer un serveur *tcp*

L'exemple qui suit met en place un serveur en utilisant le module *http*.

```
// Load the http module to create an http server.
var http = require('http');

// Configure our HTTP server to respond with Hello World to all requests.
var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});

// Listen on port 8000, IP defaults to 127.0.0.1
server.listen(8000);

// Put a friendly message on the terminal
console.log("Server running at http://127.0.0.1:8000/");
```

A noter qu'un tutoriel contenant les instructions pour l'installation et l'utilisation de *NodeJS* et *npm* est disponible en annexe.

3.4.2 Module ExpressJS

ExpressJS installé avec npm, permet de créer et gérer une application Web plus facilement parce qu'il s'occupe de création du serveur, il suffit donner une adresse ip et il le crée pour nous. Dans ce projet, il est utilisé gérer les parties accessibles depuis le serveur. L'utilisation d'ExpressJS a permis de mettre en place l'architecture décrite dans le chapitre précédent.

Exemple d'utilisation

L'exemple suivant crée un serveur local sur le port 3000. Lorsqu'un client se connecte, un message Hello World lui est transmis.

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

3.4.3 Module Socket.io

Pour ce projet, les *WebSockets* ont été choisis car ils permettent de communiquer et de partager des données avec plusieurs clients HTML. En effet, pour ce projet, le serveur Web devait recevoir et envoyer des données aux clients sans les traiter. En général l'inverse est utilisé, c'est le client qui transmet des données au serveur qui les traitent. Les *WebSockets* ont permis d'envoyer et de recevoir des données depuis le serveur sans avoir à effectuer un traitement préalable.

Socket.io est un module installé par npm permettant d'utiliser les *WebSockets*. Il permet de facilement les manipuler que ce soit du côté client ou du côté serveur. Dans ce projet, l'utilisation de *SocketIOC# Client* a nécessité d'utiliser la version 0.9 de *Socket.io*. *SocketIOC#* permet d'utiliser les *WebSockets* afin de se connecter et d'envoyer ses informations captées par le *Kinect* puis de renvoyer au client HTML. Le module *Socket.io* utilise une syntaxe précise qui doit être respectée du côté client et du côté serveur :

- lancement du serveur (seulement pour le serveur) : `io.listen(adresse ip du serveur);`
- l'attente de connexions (seulement pour le serveur) : `io.sockets.on('connection', function(socket){ réception et envoi de données});`
- connexion au serveur (seulement pour le client) : `socketIO.connect(adresse ip et port du serveur);`
- la réception de données : `socket.on(label de l'envoi, function(data){traitement des données reçue});`
- l'envoi de données : `socket.emit(label de l'envoi, données);`

Pour plus d'informations, voir [25].

Exemple d'utilisation

L'exemple suivant est un chat entre les clients connectés. Cet exemple utilise également le module *ExpressJS* parce que la transmission des messages entre les clients est gérée par un serveur. Lors de la réception d'un message, le serveur le transmet à tous les clients connectés. Cet exemple contient deux parties, un client HTML et un serveur.

```
\` Serveur
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});

\` Client HTML
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <script src="https://cdn.socket.io/socket.io-1.2.0.js"></script>
    <script src="http://code.jquery.com/jquery-1.11.1.js"></script>
  </head>
  <body>
    <ul id="messages"></ul>
    <form action="">
      <input id="m" autocomplete="off" /><button>Send</button>
    </form>
    <script>
      var socket = io();
      $('form').submit(function(){
        socket.emit('chat message', $('#m').val());
        $('#m').val('');
        return false;
      });
      socket.on('chat message', function(msg){
        $('#messages').append($('- ').text(msg));
      });
    </script>
  </body>
</html>

```

Chapitre 4

Implémentation

Ce chapitre décrit l'implémentation de l'application *Virtual-Vertigo*. Des illustrations et des extraits de codes commentés ont été insérés afin de rendre les explications plus claires.

4.1 Serveur Virtual-Vertigo

Le *serveur NodeJS* mis en place utilise les *modules ExpressJS et Socket.io*. Le *module ExpressJS* a permis de créer le serveur Web, de rediriger les clients se connectant au serveur et de rendre les fichiers nécessaires publics. Le *module Socket.io* a permis de créer un serveur utilisant les *WebSockets* avec le serveur Web créé avec le module *ExpressJS*. De plus, il permet de réceptionner les informations du *Kinect* et de les renvoyer aux clients connectés.

```
var express = require("express"), app = express(),
server = require('http').createServer(app);

// redirection on /public who content the files
app.use(express.static(__dirname + '/public'));

server.listen(3000, function(){// launch the server
  console.log('listening on *:3000');
});

//create the server socket using the server created with express
var io = require('socket.io').listen(server, { 'destroy buffer size': Infinity });
io.set('log level', 1);

//wait for the connections
io.sockets.on('connection', function(socket){
  console.info('Client connected');
  // reception of datas captured by the kinect
  socket.on("squelette", function(json){
    io.sockets.emit("dataKinect", json);
  });
  socket.on("orientation", function(start, end, rotation){
    var json = {"start": start, "end": end, "rotation": rotation};
    io.sockets.emit("boneOrientation", json);
  });
  socket.on('disconnect', function() {
    console.info('Client is gone');
  });
});
});
```

Ci-dessous un schéma illustrant l'échange de données entre le *serveur Web Virtual-Vertigo*, les clients HTML et le Kinect.

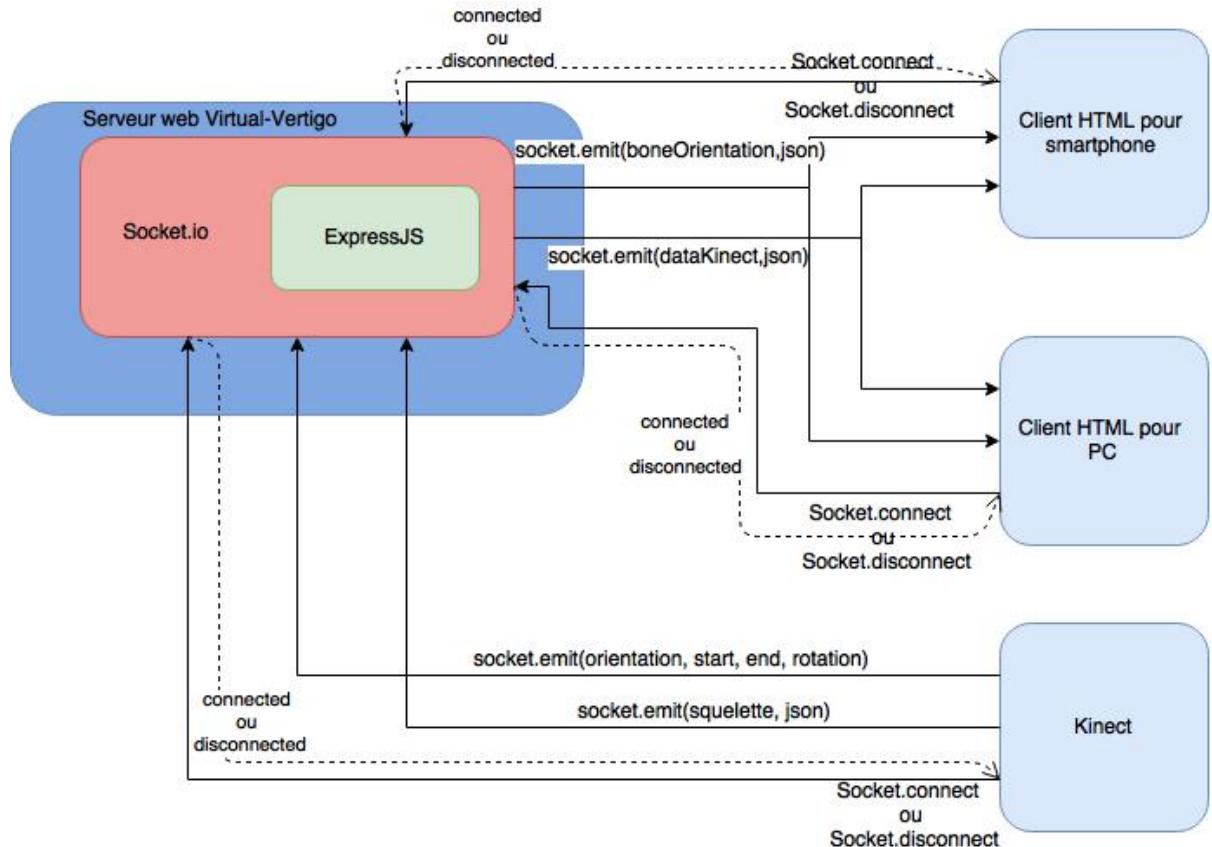


FIGURE 4.1: Illustration des échanges de données entre serveur, clients HTML et Kinect

4.2 Données provenant du Kinect

Cette section décrit comment les données provenant du *Kinect* ont été interprétées.

4.2.1 Capture des données

La capture des données du *Kinect* a été effectuée en C# avec *Visual Studio 2013*. Elle a été effectuée en plusieurs étapes :

1. La connexion au *serveur NodeJS* utilisant *SocketIOC# Client*.

```
private static bool InitializeConnection(){
    var io = new SocketIOClient();
    socket = io.Connect("http://localhost:3000/");

    if (io.Connected)
        Console.WriteLine("kinect connected to server");
    else
        Console.WriteLine("failed to connect to server");
}
```

2. L'initialisation du *Kinect*.

```
private static void InitializeKinect(){
    sensor = KinectSensor.KinectSensors.SingleOrDefault();
    if (sensor != null)
    {
        sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Default;
        sensor.SkeletonStream.Enable(new TransformSmoothParameters()
        {
            Smoothing = 0.5f,
            Correction = 0.5f,
            Prediction = 0.5f,
            JitterRadius = 0.05f,
            MaxDeviationRadius = 0.04f
        });
        sensor.AllFramesReady += Sensor_AllFramesReady;

        sensor.Start();
        Console.WriteLine("sensor started");
    }
}
```

3. La capture et l'envoi de la position de chaque articulation captée du squelette.

```
static void Sensor_AllFramesReady(object sender, AllFramesReadyEventArgs e) {
    using (var frame = e.OpenSkeletonFrame()) {
        if (frame != null){
            frame.CopySkeletonDataTo(skeletons);

            foreach (Skeleton skeleton in skeletons)
            {
                if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
                {
                    socket.Emit("squelette", skeleton.Joints);
                    [...]
                }
            }
        }
    }
}
```

4. La récupération et l'envoi de l'orientation des mains afin de pouvoir voir la rotation de la main lors de la simulation.

```
static void Sensor_AllFramesReady(object sender, AllFramesReadyEventArgs e) {
    [...]
    foreach (BoneOrientation orientation in skeleton.BoneOrientations)
    {
        if (orientation.StartJoint == JointType.WristLeft || orientation.StartJoint == JointType.WristRight)
            socket.Emit("orientation", orientation.StartJoint, orientation.EndJoint, orientation.AbsoluteRotation.Quaternion);
    }
}
```

4.2.2 Format des données

Les données de positions envoyés au serveur sont au format json et sont des valeurs de la scène réelle. Les données sont envoyées dans un tableau où chaque élément est un objet json contenant les informations :

- la valeur du point capté
- le vecteur de position
- l'état de l'articulation (captée, inférée ou non-captée), cette information n'est pas utilisée

```
Array[20]
0 : Object
  JointType: 0
  Poistion : Object
    X: valeur captée en x
    Y: valeur captée en y
    Z: valeur captée en z
  TrackingState : 0, 1 ou 2
```

FIGURE 4.2: Illustration de l'objet json des positions

Les données envoyées pour l'orientation des mains sont les suivantes :

- le point d'origine, dans notre cas, le poignet
- le point de fin, dans notre cas, la main
- le vecteur d'orientation et l'angle de rotation en radians

```
Object
  end : identifiant articulation
  start:identification de l'articulation
  rotation: Object
    W : angle de rotation
    X: valeur orientation en x
    y: valeur orientation en y
    z: valeur orientation en z
```

FIGURE 4.3: Illustration de l'objet json de l'orientation des mains

Pour plus d'informations sur les objets json, voir [26].

4.2.3 Traitement des données

Selon le diagramme de séquence 2.6, lorsque le serveur reçoit les données du Kinect, il les renvoie aux clients HTML. Lorsque les clients HTML reçoivent à leur tour les données, celles-ci sont parcourues afin de faire correspondre les valeurs reçues avec une articulation du squelette (voir table des identificateurs d'articulations 3.2). Ces données sont réceptionnées dans un nouvel objet json contenant les informations :

- le nom de l'articulation
- la position x de l'articulation
- la position y de l'articulation
- la position z de l'articulation

```
Object
Nom de l'articulation : Object
    name : nom de l'articulation
    x: valeur captée en x
    y: valeur captée en y
    z: valeur captée en z
```

FIGURE 4.4: Illustration de l'objet json créé avec les positions reçues et leurs articulations

A partir de cet objet json, le positionnement de chaque articulation peut être effectué. Au niveau des clients, un membre est représenté par deux sphères reliées par un cylindre. Afin d'adapter ces membres à la taille de chaque personne et d'animer ce personnage virtuel plusieurs étapes sont effectuées :

- le calcul de la distance entre deux articulations reliées
- le positionnement du membre entre les deux articulations
- l'adaptation de la taille du membre en fonction de la distance calculée
- le calcul de l'angle entre ses deux articulations en utilisant la fonction atan2 (voir section 4.4)
- la rotation du membre en fonction de l'angle calculé

La description détaillée de chaque étape se trouve à la section 4.4.

Finalement, le déplacement du personnage selon l'avancement de la personne est effectué en se basant sur la position de sa tête. Tout ce traitement est effectué à la réception de chaque information du *Kinect*. En effet, le *Kinect* envoie un *nouvel objet json* cycliquement. Afin de rendre le personnage réaliste, l'ajout de membres réalisistes par dessus les cylindres du personnage 3D serait envisageable.

4.3 Modélisation de réalité virtuelle

4.3.1 Crédit de la scène virtuelle

La création de la scène de réalité virtuelle a été réalisée avec *X3DOM* (voir section 3.3.1). Afin d'ajouter de la 3D sur une simple page HTML, il faut impérativement avoir les fichiers suivants :

- Le script x3dom.js permettant de manipuler la 3D
- La page de style x3dom.css afin d'utiliser la 3D

La création de la scène 3D est très simple, elle est basée sur un système de balises. Pour avoir la 3D sur la page, il y a plusieurs balises :

- la balise principale : <x3d>,
- la balise dans laquelle se trouve toute la scène 3D : <scene>,
- la balise permettant de définir de type de navigation(examine, walk, fly, etc.) : <navigationinfo>,
- la balise correspondant à la caméra : <viewpoint>,
- la balise permettant de dessiner des éléments 3D directement sur la page HTML : <shape>,
- les balises créer l'élément 3D voulu : <box>, <sphere>, <cylinder>,
- la balise permettant d'importer des scènes 3D depuis Blender en format X3D : <inline>,
- la balise permettant de modifier l'élément 3D : <transform>.

Pour en savoir plus sur les balises voir [27].

Chacune des balises utilisées doivent avoir un identifiant afin de pourvoir les manipuler. La manipulation de ces balises consiste à modifier ses attributs. L'accès aux balises s'effectue en *JQuery*. Ci-dessous un exemple d'accès à une balise via son identifiant :

```
$( "#"+name) . attr( "translation" );
```

Beaucoup de balises sont utilisées tel que <navigationInfo> afin de définir le mode examin. Cependant, seules deux balises utilisées sont modifiées au cours de la simulation.

1. La balise <transform> est utilisée pour ses attributs suivants :

- **translation** : déplace les balises se trouvant dans la balise <transform> dans la scène. Par exemple, le déplacement du personnage et de la caméra sur la planche.

```
$("#armature") . attr( "translation" , (parseFloat( position . head . z ) - 4.0) + " 7.3 0" );
$("#camera") . attr( "translation" , (parseFloat( position . head . z ) - 4.0) + " 8
" + (parseFloat( position . head . x ) - 0.2) );
```

- **rotation** : effectue une rotation sur les balises se trouvant dedans dans la scène virtuelle. Par exemple, la rotation des immeubles à 90 degrés.

```
<transform id='immeuble' translation='0 0 0' rotation='0 1 0 1.57' >
<inline url='./blender/vue_semi_finale.x3d'></inline>
</transform>
```

2. La balise <viewpoint> est utilisée pour ses attributs suivants :

- **position** : place la caméra aux coordonnées correspondantes. Par exemple, le placement initiale de la vue.

```
<Viewpoint id='vpp' DEF='viewpoint' position='0 0 5' orientation='1 0 0
-0.2' centerOfRotation='0 0 5' fieldOfView='2.0' ></Viewpoint>
```

- **centerOfRotation** : position correspondant à l'origine lors de la rotation de la vue. Par exemple, l'avancement du personnage sur la planche.

```
vp.attr("position", (parseFloat(position.head.x) -0.2) + " " + (parseFloat(
    position.head.y) -0.2)+ " " + (parseFloat(position.head.z) -0.2));
vp.attr("centerOfRotation", (parseFloat(position.head.x) -0.2) + " " + (
    parseFloat(position.head.y) -0.2)+ " " + (parseFloat(position.head.z)
    -0.2));
```

- **orientation** : quaternion correspondant à l'orientation de la caméra dans la vue. Par exemple, le déplacement lors des mouvements de la tête.

```
MYAPP.viewpoint.setAttribute("orientation", orientation[0].x + " " +
    orientation[0].y + " " + orientation[0].z + " " + orientation[1]);
```

4.3.2 Création des immeubles

Les immeubles ont été créées avec un logiciel 3D *Blender* et *Cinema4D*. Pour ce projet, deux scènes de réalité virtuelle ont été réalisées :

1. Une version basique, implémentée sur *Blender* contient 3 immeubles, une planche et le sol. Elle est illustrée sur la figure 4.5 :

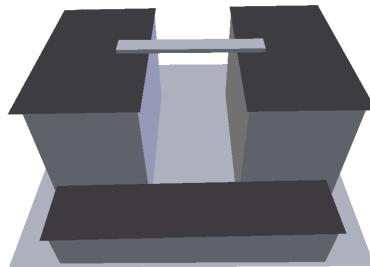


FIGURE 4.5: Modèle de vue 3D modélisé sur Blender

2. Afin de rendre la vue virtuelle plus réaliste, nous avons contacté M. OLIVIER DONZÉ, professeur de la filière architecture de paysage, et son assistant, M. BENJAMIN DUPONT-ROY, qui travaillent sur la modélisation 3D du territoire.

La deuxième version, réalisée par M. BENJAMIN DUPOND-ROY, est plus détaillé et plus réaliste. Elle a été implémentée sur *Cinema4D*. Il a modélisé la rue devant heapia en la personnalisant un peu. Voici quelques vues de son travail :



FIGURE 4.6: vue entre immeubles et du dessus

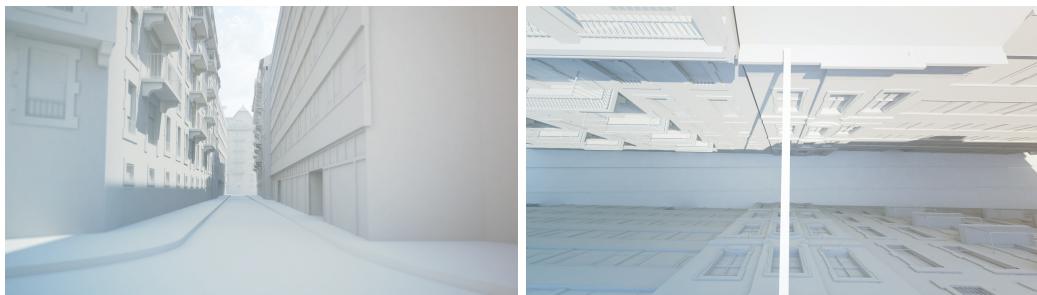


FIGURE 4.7: vue depuis la rue et depuis la planche

4.3.3 Création du personnage

La création du personnage a changé plusieurs fois lors de ce projet. Dans un premier temps, une femme avait été modélisée sur *Blender* mais n'étant pas assez réaliste, celle-ci n'a plus été utilisée.

Puis, en utilisant *MakeHuman* un homme réaliste a été créé et un squelette lui a été ajouté afin de l'animer si possible. Cet homme a été utilisé pendant une grande partie du projet mais lorsqu'il a fallu l'animer il ne convenait plus. En effet, un problème d'export sous *Blender* rendait le personnage rigide (pour plus de détails voir section 4.4).

Finalement, afin de pouvoir animer un personnage en fonction du *Kinect*, un squelette a été recréé en 3D. L'animation du personnage a été privilégiée par rapport à un rendu réaliste car priorité était de voir ses membres bouger lorsque l'on porte les *Google CardBoard*.

Ci-dessous une illustration montrant l'"évolution" du personnage 3D tout au long du projet.



FIGURE 4.8: "Evolution" du personnage 3D

4.3.4 Textures

La pose des textures, dans le cas des deux modélisations, a été effectuée à partir des logiciels 3D *Blender* et *Cinema4D*. En effet, lorsque la vue 3D est exportée en *X3D*, les textures appliquées restent sur la vue. Il faut cependant avoir à disposition les textures lors du chargement sur *X3DOM*.

Pour la vue basique à gauche, les textures ont été trouvées sur internet puis appliquées sur *Blender* et pour la vue plus réaliste à droite, une photo a été prise et extrapolée pour bien se fondre sur les immeubles.

Ci-dessous une illustration du rendu 3D avec textures au bord de la planche.

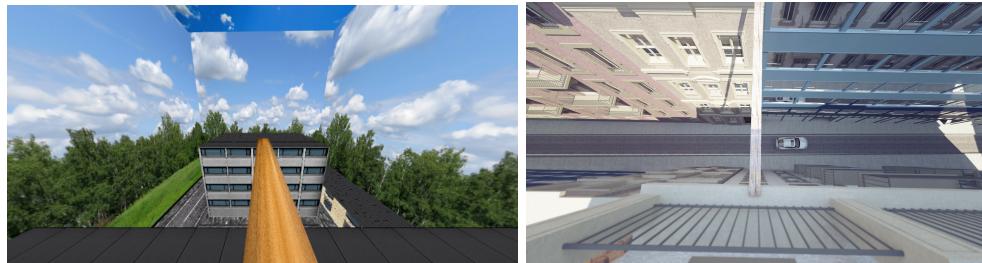


FIGURE 4.9: Illustrations vue virtuelle avec textures

4.4 Animations sur la réalité virtuelle

La prise en charge des différentes animations de la réalité virtuelle est locale au périphérique exécutant l'application.

Placement de la caméra

Le placement de la caméra s'effectue avec la balise `<viewpoint>`, en utilisant son attribut `position` correspondant au vecteur de position. Avec cette balise, il est possible de placer la caméra à la position de la tête.

Sur la vue du smartphone, la caméra se place en même temps que la tête avance. Il faut pour cela mettre à jour cet attribut à chaque changement.

Sur la vue de l'ordinateur, la caméra peut se déplacer comme sur la vue du smartphone ou être à un point fixe derrière la personne. Ces deux points de vues donnent une perspective différente.

A noter que la balise `<viewpoint>` est englobée par une balise `<transform>` qui permet d'effectuer une rotation suivi d'une translation. La rotation permet à la caméra de se placer perpendiculairement aux immeubles. La translation déplace la caméra sur un seul axe sans avoir à changer la position initiale de la caméra.

L'illustration 4.10 ci-dessous montre la position de la caméra sur le smartphone et les deux positions de caméra possibles sur l'écran standard.

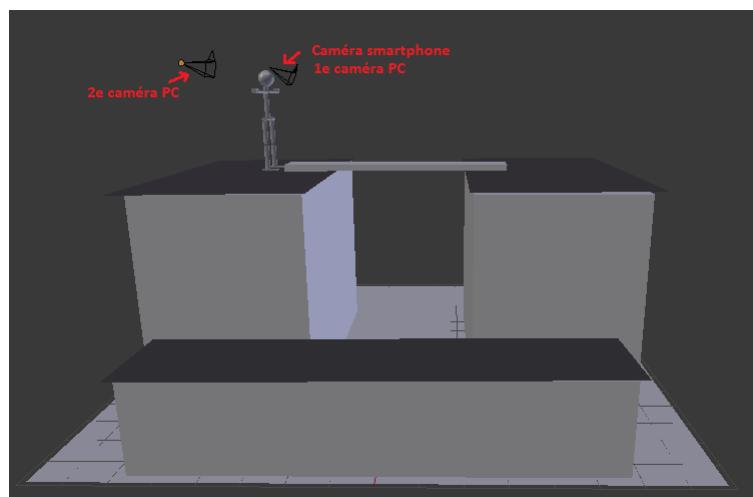


FIGURE 4.10: Illustrations des positions possibles de la caméra

Orientation de la tête

L'orientation de la tête s'effectue en utilisant l'accéléromètre du smartphone. L'accéléromètre mesure la force d'accélération appliquée au smartphone sur les axes x, y et z (force de gravitation). Il est utilisé pour détecter les mouvements. L'illustration 4.11 ci-dessous illustre les axes de l'accéléromètre.

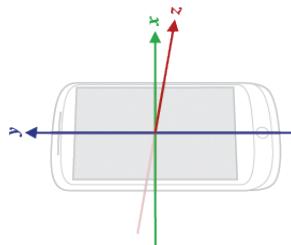


FIGURE 4.11: Illustration des axes de l'accéléromètre

La rotation de la vue en fonction des mouvements de la tête s'effectue en plusieurs étapes :

1. récupérer la rotation du smartphone dans un quaternion en utilisant des paramètres mis à disposition par *X3DOM*

```
var q0 = x3dom.fields.Quaternion.axisAngle(new x3dom.fields.SFVec3f(0,0,1),-
deg2rad(window.orientation));
```

2. créer un quaternion avec les valeurs de l'accéléromètre

```
var q = new x3dom.fields.Quaternion();
q.setFromEulerYXZ(deg2rad(MYAPP.deviceOrientation.beta), deg2rad(MYAPP.
deviceOrientation.alpha), -deg2rad(MYAPP.deviceOrientation.gamma));
```

3. créer un quaternion afin de tourner la caméra car l'orientation captée par le smartphone pointe vers le haut

```
var q1 = new x3dom.fields.Quaternion.axisAngle(new x3dom.fields.SFVec3f
(1,0,0),-Math.PI/2);
```

4. multiplier le quaternion des valeurs de l'accéléromètre avec les deux autres quaternions

```
q = q.multiply(q1);
q = q.multiply(q0);
```

5. transformer le quaternion des valeurs de l'accéléromètre en un vecteur de position avec un angle de rotation

```
var orientation = q.toAxisAngle();
```

6. appliquer les valeurs de accéléromètre à l'attribut `orientation` de la caméra

```
MYAPP.viewpoint.setAttribute("orientation", orientation[0].x + " " +
orientation[0].y + " " + orientation[0].z + " " + orientation[1]);
```

Avancement sur la planche

L'avancement du personnage sur la planche virtuelle s'est effectué en deux temps en modifiant les balises *X3DOM* :

1. Simulation du déplacement du personnage sur la planche :

Pour mettre en place ce déplacement fictif, une récursivité a été implémentée. Toutes les secondes, la distance définie diminue. Le personnage est alors déplacé et la vue suit ce déplacement. Cette récursion tourne en boucle parce que lorsque la distance est égale à la distance minimum définie, les valeurs sont réinitialisées.

```
function moveFictif() {
    timer--;
    character.attr("translation", timer +' 20 2');
    vp.attr("position",'0 10 '+ timer);
    vp.attr("centerOfRotation", '0 10 '+ timer);

    if (timer <= -MAX_Z)
        timer = MAX_Z;

    sleep(1);
    moveFictif();
}
```

2. Avancement du personnage en fonction du *Kinect* et selon la positon de la tête :

Au commencement de la simulation, la personne se place face au *Kinect* à environ 4 mètres de distance. Puis elle commence à avancer sur la planche en portant les lunettes. La vue affichée sur le *smartphone* sera adaptée en fonction de ces déplacements. Les membres peuvent être vus lors de la simulation. Pendant le déplacement, une vérification est effectuée afin de savoir si la personne a "chuté".

```
function move() {
    if (position.head != undefined)
        if ((position.head.z > 3.90 & !start) ){
            start = true;
            document.getElementById('music').play();
        } else if (position.head.z < 0.70 & start)
            start = false;

        if (start){
            // only the all armature move, the animation of the member is not
            // handle here
            vp.attr("position", (parseFloat(position.head.x) -0.2) + " " +
            parseFloat(position.head.y) -0.2)+ " " + (parseFloat(position.head.z)
            -0.2));
            vp.attr("centerOfRotation", (parseFloat(position.head.x) -0.2) +
            " " + (parseFloat(position.head.y) -0.2)+ " " + (parseFloat(position.head
            .z) -0.2));
            $("#armature").attr("translation", (parseFloat(position.head.z)
            - 4.0) + " 7.3 0");
            $("#camera").attr("translation", (parseFloat(position.head.z) -
            4.0) + " 8 " + (parseFloat(position.head.x) -0.2));
            checkBoundries();
        }
}
```

Animation du personnage

L'animation du personnage était un aspect important du projet car on voulait que la personne puisse voir ses membres lors de la simulation. Au cours du développement, l'animation du personnage a posé quelques problèmes (voir le chapitre 5). Comme mentionné dans la section 4.2.3), l'animation du personnage a été effectuée en suivant les étapes :

1. **Distance en deux dimensions entre des points $A = (A_x, A_y)$ et $B = (B_x, B_y)$:**

$$AB = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2} \quad (4.1)$$

2. **Angle entre 2 points A et B :** L'angle est calculé selon le triangle rectangle suivant :

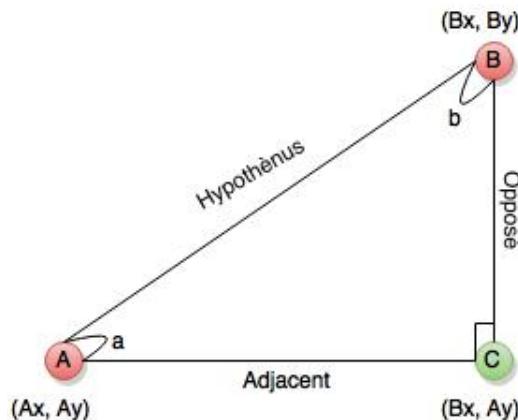


FIGURE 4.12: Triangle rectangle utilisé pour le calcul de l'angle a

$$a = \text{atan2}((B_y - A_y), (A_x - B_x)) \quad (4.2)$$

(La description de atan2 se trouve à la section 4.4).

3. **Positionnement du membre entre les points A et B :** une simple translation du cylindre correspondant à l'id à la position de l'articulation A. Le cylindre est positionné en fonction du point d'origine A et du point de fin B.

```
function positionning(id, pos){
    $("#" + id).attr("translation", " " + (pos.x) + " " + (pos.y - 0.5) + " " + pos.z)
}
```

4. **Adaptation de la taille du membre :** un scaling appliqué en x, correspondant à la longueur, au bon cylindre utilisant la distance calculée auparavant entre les 2 points A et B.

```
function scaling(id, distance, old){
    var distance = distance.toFixed(2);
    if (distance != old){
        $("#" + id).attr("scale", distance + " " + DIV_DISTANCE + " " +
        DIV_DISTANCE);
        old = distance;
    }
    return old;
}
```

5. **Rotation du membre :** une rotation est appliquée en z en utilisant l'angle calculé auparavant.

```
function anime(id, angle, old){
    var angle = angle.toFixed(2);
    if (angle != old){
        $("#" + id).attr("rotation", "0 0 1 " + angle);
        old = angle;
    }
}
```

A noter qu'un objet json a été mis en place contenant pour chaque membre relié :

- un id correspondant à l'identification du cylindre 3D
- un membre A correspondant au membre de départ
- un membre B correspondant au membre d'arrivé

Object
id : identifiant du cylindre
A : articulation de départ
B : articulation de fin

FIGURE 4.13: Illustration de l'objet json des articulations reliées

A noter également que l'ordre des membres A et B est important pour l'animation. Car s'ils sont inversés l'animation se fera dans l'autre sens. Par exemple, si le membre A est le coude gauche et le membre B l'épaule gauche, le cylindre correspondant au bras sera placé aux positions du coude. Cela affecterait également la rotation du membre car le point de départ serait le coude.

Atan2

Atan2 est une variante de la fonction arc tangente prenant en paramètres deux valeurs. Pour tout paramètre réel non nul, atan2 est l'angle en radians entre la partie positive de l'axe des x d'un plan et le point de ce plan aux coordonnées passées en paramètres. L'angle donné est positif pour les angles dans le sens trigonométrique (sens anti-horaire) et négatif dans l'autre sens (pour plus d'information sur atan2 [28]).

Définition et illustrations

Pour $y \neq 0$:

$$\text{atan2}(y, x) = \begin{cases} \varphi \cdot \text{sgn}(y) & x > 0 \\ \frac{\pi}{2} \cdot \text{sgn}(y) & x = 0 \\ (\pi - \varphi) \cdot \text{sgn}(y) & x < 0 \end{cases}$$

où φ est l'angle compris entre $[0, \Pi / 2]$ de façon à ce que $\tan(\varphi) = |y/x|$ et sgn est la fonction signe.

Et :

$$\text{atan2}(0, x) = \begin{cases} 0 & x > 0 \\ \text{non défini} & x = 0 \\ \pi & x < 0 \end{cases}$$

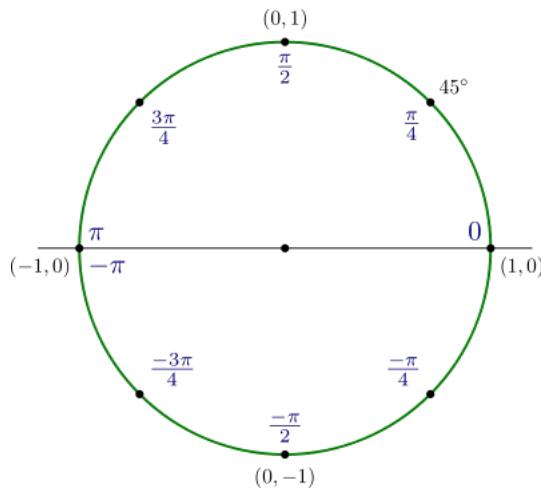


FIGURE 4.14: Illustration du cercle trigonométrique des valeurs de atan2

Le cercle ci-dessus illustre les valeurs, en radians, de atan2 . Les angles sont inscrits en bleu et les points sont inscrits à l'extérieur du cercle.

Effet sonore

L'ajout d'un effet sonore lors de la simulation semblait indispensable afin de permettre à l'utilisateur mieux s'intégrer dans sa simulation. Cet effet sonore est un enregistrement du bruit ambiant à la fenêtre. Il a été ajouté en HTML5 et est activé, en JavaScript, lorsque la personne se trouve au bord de la planche.

Chute et recommencement

Une animation de "chute" a été implémentée. Lorsque la personne "chute", c'est-à-dire qu'elle a les deux pieds en dehors de la planche. La personne voit la vue descendre au fur et à mesure de la "chute". La désactivation cette option est possible si la personne ne le supporte pas.

La possibilité de recommencer la simulation est également important afin d'éviter de devoir recharger la page à chaque fois. Lorsque la personne "chute" ou qu'elle arrive de l'autre côté de la planche, la vue de départ revient après quelques secondes et la personne peut recommencer.

Chapitre 5

Problèmes rencontrés

Comme dans chaque projet, un ensemble de problèmes se sont produits. Dans ce chapitre se trouve la description de chaque problème majeur rencontré.

Utilisation du Kinect en passant par NodeJS

A la fin du projet de semestre, un projet *GitHub Node Kinect* [29] a été trouvé. Il permet de capturer les données du *Kinect* directement avec *NodeJS*, donc de ne pas utiliser *Visual Studio* et d'avoir un projet entièrement en *JavaScript*.

Le projet *Node Kinect* utilise les outils suivants :

- La librairie *Libusb* est une librairie C, open source, qui donne accès aux périphériques USB sur beaucoup de systèmes d'exploitation (pour plus d'information voir [30]).
Elle permet de communiquer avec le *Kinect*.
- Le driver *Libfreenect* est un driver implémenté par *OpenKinect*. *OpenKinect* est une communauté open source de personnes qui veulent faire fonctionner le *Kinect* avec d'autres systèmes d'exploitation que Windows (pour plus d'informations sur *Libfreenect* [31] ou sur *OpenKinect* [32]).

Ce driver permet de capturer les données du *Kinect* et donc de substituer le *Kinect SDK*.

Lors de l'installation de *Libfreenect* et de l'installation des logiciels nécessaires à son fonctionnement, des erreurs apparaissaient car plusieurs logiciels ou librairies manquaient. Après plus d'une semaine bloquée sur cette installation et donc dans l'incapacité de commencer à utiliser le *Kinect*, l'utilisation du projet *Node Kinect* a été abandonnée. Finalement, la gestion du *Kinect* dans *Virtual-Vertigo* s'est faite avec le *Kinect SDK 1.8*, *Visual Studio* et *C#*. L'envoi des données capturées passe par un serveur Web.

Utilisation KinectHTML5 sur un site sécurisé

Toujours durant le travail de semestre, le projet *Visual Studio KinectHTML5* [33] a été trouvé. Ce projet permet de capturer des données du *Kinect* et de les envoyer via les *WebSockets* à un client *HTML5*. Il a été choisi car il correspondait aux besoins du projet *Virtual-Vertigo*. Le projet *KinectHTML5* est constitué de deux parties :

1. un projet client dans lequel se trouve la page HTML5 et les scripts nécessaires à la connexion aux *WebSockets*, à la récupération des données du *Kinect* et l'affichage des informations captées sur un canvas;
2. un projet serveur sur lequel s'exécute le serveur *WebSocket* et où se font la capture et l'envoi des données.

En utilisant ce projet localement tout fonctionnait parfaitement. Cependant, lors de l'intégration de ce projet au serveur d'hébergement, ça ne fonctionnait plus. En effet, *KinectHTML5* utilise des *WebSockets* tandis que le serveur d'hébergement utilise HTTPS. Pour que le projet fonctionne, il fallait l'adapter pour utiliser des *WebSockets* sécurisés. Mais les *WebSockets* sécurisés nécessitent un certificat. Celui-ci a alors été créé mais il a été impossible de l'ajouter au serveur d'hébergement.

La solution a été d'utiliser un *serveur NodeJS* avec *Socket.io* servant de pont entre les clients HTML et le *Kinect*. Après quelques recherches, le *projet Visual Studio SocketIOC# Client* a été trouvé. Il permet à un programme C# de se connecter à un *serveur NodeJS* utilisant *Socket.io* (pour plus d'informations sur le projet *SocketIOC#* voir [17] ou sur *Socket.io* [25]).

Animation du personnage - mapping squelette kinect et squelette blender

Lorsque le personnage virtuel 3D a été créé avec *MakeHuman*, un squelette a été ajouté au personnage afin de pouvoir l'animer. Pour une animation la plus réaliste possible, l'idée de mapper le squelette du *Kinect* et le squelette présent dans le personnage 3D est apparue. Cependant, après quelques recherches et tests, il était impossible de mapper les deux squelettes. En effet, lorsque le personnage est exporté en format X3D avec *Blender*, le personnage devient rigide. Il est impossible d'accéder à chaque membre du squelette dans le personnage.

La solution mise en place favorise l'animation des membres au dépend du réalisme du personnage 3D. Le squelette de la personne a été reconstruit en 3D directement sur *X3DOM*. En utilisant les données reçues, cela a permis d'animer les membres du personnage. L'animation des membres a été effectuée en calculant l'angle entre les articulations reliées. Cet angle permet de savoir comment effectuer la rotation du membre.

Animation du personnage - calcul de l'angle

Comme mentionné dans la section précédente, un angle a été calculé entre les articulations reliées. L'angle a été calculé, considérant chaque articulation comme un point dans l'espace. Le calcul de l'angle entre deux points est impossible. Nous avons donc calculé un troisième point avec les coordonnées des deux autres points afin de créer un triangle rectangle.

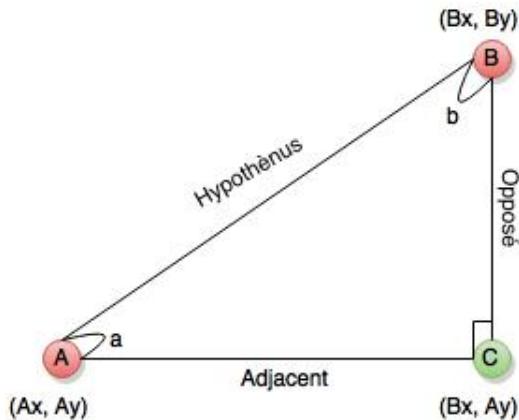


FIGURE 5.1: Illustration du triangle rectangle

Sur l'illustration 5.1, les deux points correspondant aux articulations sont en rouge, le troisième point étant en vert (une combinaison des deux autres points). En traçant les droites entre chaque point, il est possible de créer un triangle rectangle. Il permet d'utiliser les formules trigonométriques, telles que le calcul de la distance entre deux points et le calcul des angles dans un triangle rectangle.

Les formules de calcul de distance entre deux points appliquées sur ce triangle sont les suivantes :

$$AC = \sqrt{(B_x - A_x)^2 + (A_y - A_y)^2} \quad (5.1)$$

$$BC = \sqrt{(B_x - B_x)^2 + (A_y - B_y)^2} \quad (5.2)$$

$$AB = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2} \quad (5.3)$$

où $A = (A_x, A_y)$, $B = (B_x, B_y)$ et $C = (B_x, A_y)$

Les formules de calcul de l'angle appliquées sur ce triangle sont les suivantes :

$$b = \arccos(BC/AB) \quad (5.4)$$

$$a = \arcsin(AC/AB) \quad (5.5)$$

Cependant, en effectuant l'implémentation de ces calculs, ceux-ci n'étaient pas assez précis et trop lents, parce que la disposition des articulations n'a pas été prise en compte. Les membres ne se déplaçaient pas comme ils le devaient. Finalement, le calcul de l'angle a été effectué en utilisant la fonction atan2 décrite à la section 4.4. L'utilisation de cette fonction a rendu l'animation des membres plus précise et dynamique.

Des données captées à la réalité

Lors du développement, nous avons réalisé que le *Kinect* ne transmettait pas les données qu'il capte en fonction de la réalité. En effet, le *Kinect* transmettait les données en fonction de son angle de vue conique. En effet l'utilisation d'un objet **CoordinateMapper** ne renvoyait pas les dimensions réelles de la personne. Pour remédier à ce problème, des mesures ont été effectuées sur un rectangle délimité :

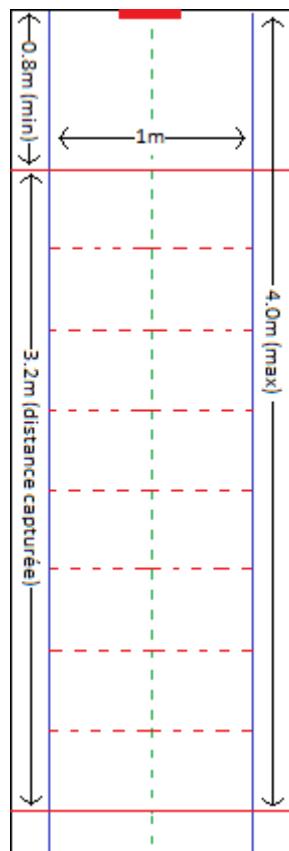


FIGURE 5.2: Rectangle créé sur le sol afin d'effectuer les mesures

Ces mesures ont été réalisées tous les 40 centimètres au centre du rectangle. Les points de repères suivants ont été définis :

- pour la distance entre deux articulations, le centre des épaules et la colonne vertébrale
- pour l'axe x, la main posée sur une table à 3 endroits différents afin ne pas fausser les mesures entre chaque déplacement
- pour z, les 2 pieds placés après les lignes tracées au sol

Voici les résultats obtenus :

Z	Distance	Z Kinect	Coin gauche	Milieu	Coin droit
0.80	169	0.85	44	276	475
1.20	147	1.25	72	271	447
1.60	112	1.65	105	294	420
2.00	92	2.05	134	274	397
2.40	79	2.46	158	261	362
2.80	69	2.86	160	257	357
3.20	60	3.24	177	270	343
3.60	54	3.65	183	257	334
3.85	50	3.90	187	302	290

En analysant ce tableau, les valeurs en z correspondent à la réalité moyennant un décalage de 0.5 cm. Cependant les valeurs mesurées en x ne correspondent pas et n'aide pas à trouver une formule pouvant être appliquée. Un graphique peut être tracé avec les valeurs des distances entre le centre des épaules et la colonne vertébrale :

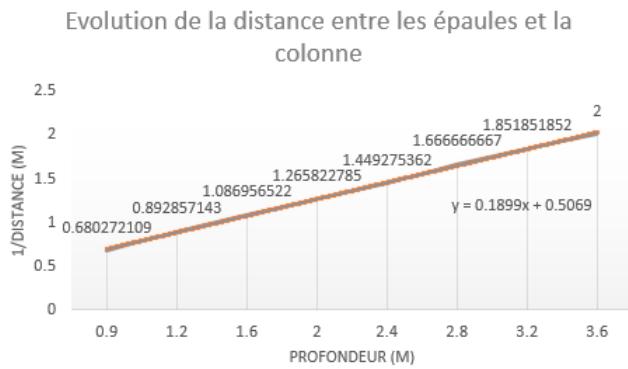


FIGURE 5.3: Évolution de la distance entre les épaules et la colonne

Finalement, le problème a été corrigé sans utiliser ces mesures. Le programme capturant les données du *Kinect* a complètement été réécrit afin de capturer les données de manière absolue. En utilisant cette méthode, les valeurs en x et y capturées sont en mètres et correspondent à la réalité.

Taille de la scène finale

Comme mentionné dans le chapitre 2.3, une des vues finales du projet a été modélisée par M. BENJAMIN DUPOND-ROY. Malheureusement, la première vue modélisée était trop détaillée et le fichier correspondant trop grand ce qui empêchait le smartphone de l'afficher. Le chargement sur PC était également ralenti. M. BENJAMIN DUPOND-ROY a donc modélisé une vue plus petite afin qu'elle s'affiche correctement sur le smartphone et ne ralentisse pas le PC.

Ci-dessous un tableau illustrant la taille des fichiers X3D générés par *Blender* et utilisés sur *X3DOM* :

Scènes	Taille fichier X3D en Ko
Scène basique contenant les textures	28 Ko
Première scène modélisée par M. BENJAMIN DUPOND-ROY sans textures	37 505 Ko
Deuxième scène modélisée par M. BENJAMIN DUPOND-ROY sans textures	11 396 Ko

TABLE 5.1: Tableau illustrant les tailles des scènes modélisées exportées en X3D

Comme vous pouvez le voir, la scène basique est très petite. Cela permet d'être chargée rapide et d'avoir une simulation fluide. La deuxième scène modélisée par M. BENJAMIN DUPOND-ROY est trois fois plus petite que la première. Elle s'affiche correctement sur le smartphone et le PC mais n'est pas aussi fluide que la scène basique.

Calibrage de la vue sur le smartphone

Lorsque la personne se trouve au bord de la planche prête à commencer la simulation, le smartphone prend en charge l'orientation de la vue en utilisant son accéléromètre. Parfois, lorsqu'il prend la main, le smartphone décale la vue. Elle n'est plus alignée sur la planche ce qui fait que la personne sur la planche se décale.

Deux solutions pourraient être implémentées :

1. utiliser l'aimant des *Google CardBoard* pour interagir avec le smartphone afin de déclencher un événement de calibrage de la vue;
2. utiliser un signe de la main que le *Kinect v1* reconnaît afin de signaler que le calibrage doit s'effectuer.

Chapitre 6

Conclusion

6.1 Bilan

L'objectif du projet *Virtual-Vertigo* était de réaliser un exercice pour combattre la peur du vide en utilisant des *Google CardBoard* pour visualiser un déplacement dans un monde virtuel. Celui-ci est constitué d'une planche reliant deux immeubles. La personne doit traverser cette planche pour atteindre l'immeuble d'en face sans "chuter".

Les éléments suivants ont été mis en place afin d'avoir une simulation opérationnelle :

- La modélisation de deux scènes virtuelles 3D constituées d'une planche reliant deux immeubles.
Cette scène est affichée sur une page Web en utilisant le *Framework X3DOM* permettant d'afficher des scènes 3D. Les scènes ont été modélisées en utilisant des logiciels 3D *Blender* et *Cinema4D*.



FIGURE 6.1: Illustration des scènes virtuelles 3D créées

- La mise en place du *serveur Web Virtual-Vertigo* servant de pont entre le *Kinect version 1* et les clients HTML (le smartphone et le PC).
Ce serveur utilise *NodeJS* avec les modules *ExpressJS* et *Socket.io*.
- La création d'un *projet Visual Studio* pour le *Kinect* pour la capture des données de positions (articulations du squelette et orientations des mains) et de les transmettre au *serveur Web Virtual-Vertigo* en utilisant *SocketIOC#*.

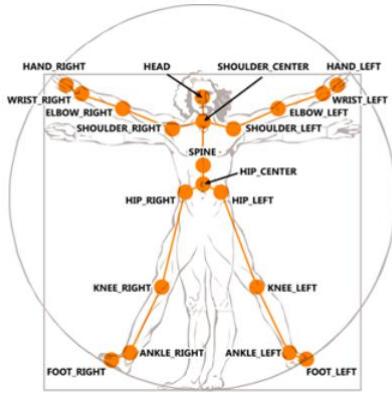


FIGURE 6.2: Illustration des articulations du squelette capturées par le *Kinect v1*

- La création d'un personnage 3D réaliste avec *MakeHuman*.
Ce personnage n'est pas utilisé dans la version finale car il est devenu rigide lors de l'exportation en X3D depuis *Blender*.

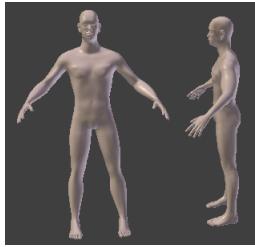


FIGURE 6.3: Illustration du personnage réaliste créé

- La création d'un client HTML pour le smartphone contenant une vue stéréoscopique de la réalité virtuelle utilisant *X3DOM*.
Un problème avec le positionnement des textures visible sur la figure 6.4 persiste.

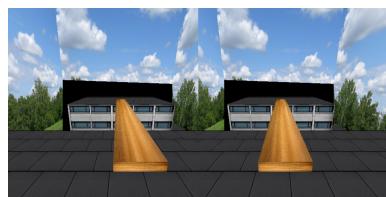


FIGURE 6.4: Illustration de la vue stéréoscopique sur le smartphone

- La création d'un client HTML pour le PC pour afficher la simulation sur un écran standard toujours avec *X3DOM*.
- La construction d'un squelette 3D dans la réalité virtuelle afin de l'animer.
Des sphères sont placées sur les articulations et des cylindres relient les articulations voisines.

- Le déplacement (en JavaScript) de la personne sur la planche dans la réalité virtuelle en fonction de la position de la tête capturée par le *Kinect version 1*.



FIGURE 6.5: Illustration du déplacement du squelette 3D dans la réalité virtuelle

- L’animation des membres du squelette 3D en fonction des données capturée par le *Kinect version 1*.
Cela nécessitait de positionner et calculer l’angle entre deux articulations voisines du squelette.
- L’adaptation de la scène virtuelle en fonction de l’orientation de la tête.
Cette orientation est détectée via les accéléromètres du smartphone.
- L’ajout d’effets afin de rendre la simulation plus réaliste :
 - une planche réelle sur le sol.
 - un bruit ambiant en utilisant un casque sans fil.
 - une animation de "chute" lorsque l’utilisateur n’a plus les pieds sur la planche.
La simulation recommence après une "chute" ou lorsque l’utilisateur est arrivé à l’extrême de la planche.

La figure 6.6 montre à gauche un camarade de diplôme, M. VASCO MARQUES, effectuant la simulation du projet et à droite la vue affichée sur les *Google CardBord* lors de sa simulation.

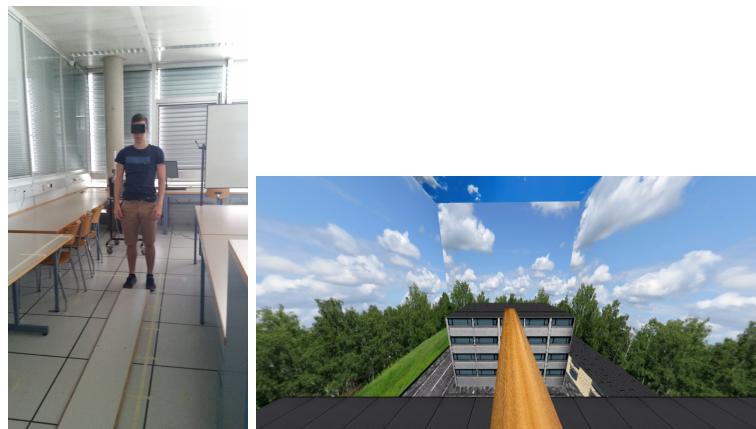


FIGURE 6.6: Simulation de Virtual-Vertigo

A la fin de ce travail de Bachelor, une simulation complète est opérationnelle. Il est possible de porter les *Google CardBoard* et de faire la simulation en ayant une planche posée sur le sol et un casque bluetooth diffusant un bruit ambiant enregistré. Cependant quelques problèmes n’affaiblissent pas la simulation persistent :

1. le décalage de la vue lorsque le smartphone prend le relais pour l'orientation de la vue en fonction de la tête;
2. le positionnement non optimal des textures dans la vue affichée sur le smartphone;
3. la visualisation non optimale des membres qui parfois ne s'affichent pas alors que la personne tient ses mains bien devant elle (ils paraissent plus petits en virtuel que dans la réalité).

6.2 Améliorations

Quelques améliorations possibles pour ce projet sont les suivantes :

- corriger les problèmes persistants
- ajouter le reste des effets de réalisme (la montre connectée et le ventilateur)
- utiliser la *Kinect v2* afin d'avoir plus de précisions sur l'orientation des mains
- utiliser le smartphone pour lancer l'effet sonore au lieu du casque sans fil
- trouver d'autres effets de réalisme pour augmenter l'immersion de la personne
- améliorer les *Google CardBoard* afin de les rendre plus esthétiques et plus agréable à porter
- étudier la possibilité d'un composant disponible sur le smartphone en remplacement de la *Kinect*

6.3 Perspectives

Afin de pouvoir vraiment utiliser le projet *Virtual-Vertigo* dans un contexte médical, il faudrait contacter un psychologue ou un spécialiste pour évaluer le projet. Cette évaluation permettrait de savoir si l'approche choisie pour ce projet est vraiment utile pour aider les personnes ayant le vertige. En cas de succès auprès d'un spécialiste, le projet pourrait être lancé avec une startup et serait distribué dans la médecine thérapeutique. Une autre possibilité de vente serait envisageable. En effet, l'armée ou les services de pompiers pourraient être intéressés par des variantes de *Virtual-Vertigo*. Celles-ci pourraient leur être utile afin de tester ou entraîner leurs nouvelles recrues dans un environnement virtuel similaire à celui de *Virtual-Vertigo*.

Annexe A

Glossaire

Réalité virtuelle

Simulation informatique interactive immersive, en temps réel pouvant être visuelle, sonore ou haptique d'environnements réels ou imaginaires.

Google CardBoard

Casque de réalité virtuelle en carton contenant des lentilles et un smartphone.

Stéréoscopique

Technique mise en oeuvre pour reproduire une perception de 3D à partir de deux images.

Kinect version 1

Périphérique initialement destiné à la console de jeux Xbox 360. Il permet à l'utilisateur d'interagir en utilisant son corps plutôt qu'une manette.

X3DOM

Framework open source permettant d'exécuter des scènes 3D sur une page Web. Il est la composition entre X3D (Extensible 3D Graphics) et DOM (Document Object Model).

X3D

Format de fichier graphique et multimédia orienté 3D utilisé dans *X3DOM* pour l'ajout d'une scène 3D modélisée sur *Blender*.

DOM

Standard du W3C qui décrit une interface indépendante de tout langage de programmation et de toute plate-forme, permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents HTML et XML.

NodeJS

Framework logiciel et évènementiel en JavaScript. Il contient une bibliothèque de serveur HTTP, ce qui permet de faire tourner et de mieux contrôler un serveur Web sans avoir besoin d'un logiciel externe.

Socket.io

Module de NodeJS permettant d'utiliser les *WebSockets*. Il permet de facilement les manipuler que ce soit du côté client ou du côté serveur.

ExpressJS

Module de NodeJS permettant de créer et gérer une application Web plus facilement parce qu'il s'occupe de création du serveur, il suffit donner une adresse ip et il le crée pour nous.

SocketIOC#

Projet disponible sur *GitHub* permettant de faire communiquer un programme C# avec un serveur NodeJS utilisant *Socket.io*.

Blender

Logiciel de modélisation, d'animation et de rendu 3D développé par la Fondation Blender.

Cinema4D

Logiciel libre de modélisation 3D de corps humains.

MakeHuman

Logiciel de création 3D développé par Maxon permettant la modélisation, le texturage, l'animation et le rendu d'objets 3D.

Annexe B

Cahier de bord

Afin d'avoir un suivi durant le travail de Bachelor, un cahier de bord a été mis en place. Il était mis à jour tous les jours en notant le travail effectué, les problèmes rencontrés et des informations supplémentaires si nécessaires.

Jour	Tâches	Bugs rencontrés	Infos
Semaine 1	17.04.2015 Rendez-vous initial bachelor		
	21.04.2015 Préparation de la documentation, complément du planning et listage des améliorations possibles		
	22.04.2015 Installation logiciel NodeKinect (ainsi que les dépendances), recherche placement caméra fixe et rendez vous Mr 3D	Utilisation du Kinect en passant pas NodeJS	
	23.04.2015 Compilation libfreenect, stabilisation vue, recherche placement caméra, Rendez-vous d'informations	Utilisation du Kinect en passant pas NodeJS	libfreenect est utilisé afin de pouvoir installer node-kinect.
	24.04.2015 Compilation libfreenect, rechercher et test déplacement vue, visite à Mr Regamey	Utilisation du Kinect en passant pas NodeJS	Mr Regamey pas là.
Semaine 2	27.04.2015 Compilation libfreenect, ajout du personnage, déplacement fictif du personnage (index3.html)	Utilisation du Kinect en passant pas NodeJS	
	28.04.2015 Recherche sur format des données que retourne le kinect, reinstallation sdk kinect v1.8, test placement canvas sur x3dom, récupération de la planche	Utilisation du Kinect en passant pas NodeJS	Correction du bug - Je vais laisser libfreenect de côté pour le moment, je vais utiliser KinectHTML5 (idée : placer un canvas par-dessus la balise x3dom afin de pouvoir y reporter ce que capte le Kinect)
	29.04.2015 Test kinect HTML localement, placement canvas et rendez-vous hebdomadaire		
	30.04.2015 Test capter données kinect sur git.hepia.ovh, placement de la caméra sur la tête du personnage	Utilisation KinectHTML5 sur site sécurisé	
01.05.2015	CONGÉ		
Semaine 3	04.05.2015 Mise en place du socketio client C#	Utilisation KinectHTML5 sur site sécurisé	Correction du bug
	05.05.2015 Communication client C#, serveur node et client html, test sur le smartphone, création des urls , rendez-vous Albuquerque		

Semaine 3	06.05.2015	Création du personnage 3D avec makehuman, placement de la caméra, recherche déplacement du personnage sur blender avec la kinect		
	07.05.2015	Placement de la caméra au niveau de la tête du personnage et adaptation de la rotation		
	08.05.2015	Stabilisation/adaptation de la rotation, rendez-vous hebdomadaire, création squelette rapport		
Semaine 4	11.05.2015	Positionnement caméra et rotation de la vue en fonction de la tête, recherche sur kinect		
	12.05.2015	Recherche sur kinect, déplacement sur la planche en fonction des données du kinect, recherche mapping squelette blender et squelette kinect	Animation du personnage - mapping squelette kinect et squelette blender	
	13.05.2015	Recherche + test mapping squelette blender et squelette kinect et changement position déplacement personnage	Animation du personnage - mapping squelette kinect et squelette blender	
	14.05.2015	CONGÉ		
	15.05.2015	Rapport laTex et optimisation déplacement sur la planche		
Semaine 5	18.05.2015	ABSENCE		
	19.05.2015	Rappel trigonométrie triangle rectangles et test application (elbow et shoulder) sur x3dom		
	20.05.2015	Reconstruction/animation du bras, rendez-vous hebdomadaire et correction de l'angle	Animation du personnage - calcul de l'angle	Correction du bug
	21.05.2015	Positionnement du bras au centre, reconstruction/animation du bras, reconstitution squelette X3DOM + animation squelette (articulation et membres)		bord du "bras" sur origine et non le centre
	22.05.2015	Rercherche + test orientation main et documentation		BoneOrientation mais ne marche pas bien. J'ai rechercher avec la profondeur mais pas trouver

		CONGÉ	
Semaine 6	25.05.2015		
	26.05.2015	Rercherche + test orientation main et documentation	
	27.05.2015	Optimisation déplacement + articulation personnages, recherche orientation main et documentation	
	28.05.2015	Adaptation déplacement + articulation sur smartphone	Des données captées à la réalité
	29.05.2015	Adaptation taille personnage en fonction z et rendez-vous hebdomadaire	Des données captées à la réalité
Semaine 7	01.06.2015	Rapport et test intégration vue de Benjamin	Taille de la scène finale J'ai du enlever des détails afin de pouvoir afficher le vue sur le smartphone car il était trop lourd
	02.06.2015	Rapport et test intégration vue de Benjamin	Taille de la scène finale
	03.06.2015	Rapport et test intégration vue de Benjamin	Taille de la scène finale
	04.06.2015	Rechercher solution pour scaling en fonction de z et positionnement vue 3D	Des données captées à la réalité
	05.06.2015	Rechercher solution pour scaling en fonction de z,positionnement vue 3D + ajout personnage et rendez-vous hebdomadaire	Des données captées à la réalité
Semaine 8	08.06.2015	Création du rectangle afin d'effectuer les mesures pour mapping réalité et kinect, explications dans le rapport et continuation du rapport	Des données captées à la réalité
	09.06.2015	1er prise des mesures en z et rapport	Des données captées à la réalité
	10.06.2015	2e mesures en z, mesures en x pour mapping et rapport	Des données captées à la réalité

Semaine 8	11.06.2015	Adaption de la vue final pour smartphone, Rendez-vous hebdomadaire et test textures sur blender	Taille de la scène finale	
	12.06.2015	Pose des textures sur la vue initiale, test complet smartphone	Taille de la scène finale	
Semaine 9	15.06.2015	Résolution problème scaling du personnage en fonciton de z et rechercher sur boneorientation	Des données captées à la réalité	"Correction" du bug - concentration sur une vue plus simple
	16.06.2015	Positionnement du personnage et de caméra sur la scene du pc et rapport		
Semaine 10	17.06.2015	Adaptation vue pour scene smartphone, test fonctionnel avec les cardboard et rapport		Reste quelques problèmes dûs à l'accélération
	18.06.2015	Rapport et tri		
Semaine 11	19.06.2015	Vérification de la mort du personnage et simulation de la chute, traitement de l'orientation des mains et rapport		
	22.06.2015	Recherche et test pour ajout effet sonore		
Semaine 12	23.06.2015	Rapport et orientation de la main		
	24.06.2015	Rapport		
Semaine 11	25.06.2015	Rapport et rendez-vous hebdomadaire		
	26.06.2015	Rapport		
Semaine 12	29.06.2015	Rapport		
	30.06.2015	Rapport et correction		
Semaine 12	01.07.2015	Rapport et correction		
	02.07.2015	Rapport et correction		
Semaine 12	03.07.2015	Rapport et correction		
	06.07.2015	Rapport et correction		
Semaine 12	07.07.2015	Rapport et correction		
	08.07.2015	RENDU RAPPORT		

Annexe C

Planning

Au début du travail de Bachelor, un planning initial a été créé afin de séparer le projet *Virtual-Vertigo* en tâches distinctes et d'avoir des délais à tenir. Un planning au cours du travail de Bachelor a également été créé afin de pouvoir savoir s'il y a du retard sur les tâches définies dans le planning initial.

Planning initial

Planning final

Annexe D

Code source

Le code source est disponible sur un CD annexé à ce rapport. Cependant quelques extraits sont fournis ci-dessous.

Le fichier HTML de la scène virtuelle du smartphone :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Vertigo smartphone</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <link rel='stylesheet' type='text/css' href='x3dom/x3dom.css'></link>
    <script src="http://code.jquery.com/jquery-1.6.2.min.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src='x3dom/x3dom.js'></script>
    <script src='./script.js'></script>
    <style>
      body { margin: 0px; overflow: hidden; }
      x3d{ width: 100%; height: 100%; }
    </style>
  </head>
  <body>
    <div onclick="fullscreen () ;"><h1>Fullscreen</h1></div>
    <p>Status: <label id="status">None</label></p>
    <audio id="music" preload="auto">
      <source src="music.mp3" type="audio/mp3">
    </audio>

    <x3d id="x3d">
      <scene id="scene">
        <!-- define the camera and the type of naviagtion-->
        <NavigationInfo headlight="false" type='EXAMINE'></NavigationInfo>
        <transform DEF='camera' id='camera' translation='-2 7 0'
          rotation='0 1 0 1.57'>
          <viewpoint id='vpp' DEF='viewpoint' position='0 0 4'
            centerOfRotation='0 0 5' orientation='1 0 0 -0.2'
            fieldOfView="3.1" zNear="0.001"></viewpoint>
          <viewpoint DEF='AOPT_CAM' position="0 0 5" centerOfRotation="0 0 5"/>
        </transform>

        <!-- creation of the scene -->
        <group id='unrendered_scene' render='false'>
          <group DEF='scene'>
            <!-- Immeuble -->
            <transform DEF='camera' id='camera' translation='0 0 0'>
```

ANNEXE D. CODE SOURCE

```
        rotation='0 1 0 1.57'>
<inline nameSpaceName="immeuble" mapDEFToID="true"
       url="./blender/vue_semi_finale.x3d"></inline>
</transform>

<!-- construction du squelette du personnage -->
<transform id='armature' translation='0 7.3 0' rotation='0 1 0 1.57' >
<group id='armatureGroup'>
<!-- joints -->
<transform id='HESC' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='SCSR' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='SER' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='EWR' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='WHR' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='SCSL' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='SEL' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='EWL' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='WHL' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='SCS' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='SHC' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='HCHR' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='HKR' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
<Inline url="./blender/membre.x3d"/>
</transform>
<transform id='KAR' scale='0.2 0.2 0.2' center="0 0.5 0"
          translation="0 -0.5 -0.5">
</transform>
```

```

        translation="0 -0.5 -0.5">
    <Inline url="./blender/membre.x3d"/>
</transform>
<transform id='AFR' scale='0.2 0.2 0.2' center="0 0.5 0"
    translation="0 -0.5 -0.5">
    <Inline url="./blender/membre.x3d"/>
</transform>
<transform id='HCFL' scale='0.2 0.2 0.2' center="0 0.5 0"
    translation="0 -0.5 -0.5">
    <Inline url="./blender/membre.x3d"/>
</transform>
<transform id='HKL' scale='0.2 0.2 0.2' center="0 0.5 0"
    translation="0 -0.5 -0.5">
    <Inline url="./blender/membre.x3d"/>
</transform>
<transform id='KAL' scale='0.2 0.2 0.2' center="0 0.5 0"
    translation="0 -0.5 -0.5">
    <Inline url="./blender/membre.x3d"/>
</transform>
<transform id='AFL' scale='0.2 0.2 0.2' center="0 0.5 0"
    translation="0 -0.5 -0.5">
    <Inline url="./blender/membre.x3d"/>
</transform>

<!-- members-->
<transform id='head '><Shape>
    <Sphere radius="0.1" />
</Shape> </transform>
<transform id='shouldercenter '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='shoulderleft '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='elbowleft '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='wristleft '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='handleft '><Shape>
    <Sphere radius="0.07" /> </Shape>
<!-- thumb -->
<transform id='thumbleft' translation="0.1 0 0">
    <Shape><Cylinder height="0.03" radius="0.03" /></Shape>
</transform>
<transform id='shoulderright '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='elbowright '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='wristright '><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='handright '><Shape>
    <Sphere radius="0.07" /> </Shape>
<!-- thumb -->
<transform id="thumbbright" translation="-0.1 0 0">
    <Shape><Cylinder height="0.03" radius="0.03" /></Shape>
</transform>
```

```

</transform>
<transform id='spine'> <Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='hipcenter'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='hipleft'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='kneyleft'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='ankleleft'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='footleft'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='hipright'><Shape>
    <Sphere radius="0.07" />
</Shape></transform>
<transform id='kneeright'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='ankleright'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
<transform id='footright'><Shape>
    <Sphere radius="0.07" />
</Shape> </transform>
</group>
</transform>
</group>
</group>

<!-- creation of the left view -->
<group DEF='left' render='true' class='vue'>
    <shape>
        <appearance>
            <renderedTexture id="left" stereoMode='LEFT_EYE' update='ALWAYS'
                interpupillaryDistance='0.3' dimensions='800 800 4'
                repeatS='false' repeatT='false'>
                <viewpoint USE='viewpoint' containerField='viewpoint'></viewpoint>
                <group USE='scene' containerField="scene"></group>
            </renderedTexture>
            <composedShader>
                <shaderPart type='VERTEX'>
                    attribute vec3 position;
                    attribute vec2 texcoord;
                    uniform mat4 modelViewProjectionMatrix;
                    varying vec2 fragTexCoord;
                    void main()
                    {
                        vec2 pos = sign(position.xy);
                        fragTexCoord = texcoord;
                        gl_Position = vec4((pos.x/2.0)-0.5, pos.y, 0.0, 1.0);
                    }
                </shaderPart>
                <shaderPart DEF="frag" type='FRAGMENT'>
                    #ifdef GL_ES
                    precision highp float;

```

```

#ifndef
uniform sampler2D tex;
uniform float leftEye;
varying vec2 fragTexCoord;
void main()
{
    gl_FragColor = texture2D(tex, fragTexCoord);
}
</shaderPart>
</composedShader>
</appearance>
<plane solid="false"></plane>
</shape>
</group>

<!-- creation of the right view -->
<group DEF='right' render='true' class='vue'>
<shape>
<appearance>
<renderedTexture id="right" stereoMode='RIGHT_EYE' update='ALWAYS'
    interpupillaryDistance='0.3' dimensions='800 800 4'
    repeatS='false' repeatT='false'>
    <viewpoint USE='viewpoint' containerField='viewpoint'></viewpoint>
    <group USE='scene' containerField="scene"></group>
</renderedTexture>
<composedShader>
<shaderPart type='VERTEX'>
    attribute vec3 position;
    attribute vec2 texcoord;
    uniform mat4 modelViewProjectionMatrix;
    varying vec2 fragTexCoord;
    void main()
    {
        vec2 pos = sign(position.xy);
        fragTexCoord = texcoord;
        gl_Position = vec4((pos.x + 1.0) / 2.0, pos.y, 0.0, 1.0);
    }
</shaderPart>
<shaderPart USE="frag" type='FRAGMENT'>
</shaderPart>
</composedShader>
</appearance>
<plane solid="false"></plane>
</shape>
</group>
</scene>
</x3d>
</body>
</html>

```

La récupération des données reçues par le *Kinect* en JavaScript :

```

window.onload = function () {
    var character = $("#character");
    var vp = $("#vpp");
    var status = $("#status");
    status.text("Connecting to server...");

    /* getting the informations of the kinect */
    // Connect to server.
    var socket = io.connect('http://'+ ip +':3000');
    status.text("Connection successful.");

```

```

// Receive data FROM the server!
socket.on("dataKinect", function(json){
    status.text("Kinect data received.");
    console.log(json);

    for (var i = 0; i < json.length; i++){
        for (var j=0; j < jointsType.length; j++){
            // map the joint type to his name
            if (json[i].JointType == j)
                position[jointsType[j].joint] =
                {"name" : jointsType[j].joint ,
                 "x" : json[i].Position.X.toFixed(3) * REAL,
                 "y" : json[i].Position.Y.toFixed(3) * REAL,
                 "z" : json[i].Position.Z.toFixed(3)};
        }
    }
    animate(position);
});

socket.on("boneOrientation", function(boneOrientation){
    //var startjoint = jointsType[boneOrientation.start].joint;
    var endjoint = jointsType[boneOrientation.end].joint;

    // rotation of the thumb
    if (endjoint == 'handleft')
        $("#thumbleft").attr("rotation",
            boneOrientation.rotation.X.toFixed(3) +
            " " + boneOrientation.rotation.Y.toFixed(3) +
            " " + boneOrientation.rotation.Z.toFixed(3) +
            " " + boneOrientation.rotation.W.toFixed(3));
    else if (endjoint == 'handright')
        $("#thumbright").attr("rotation",
            boneOrientation.rotation.X.toFixed(3) +
            " " + boneOrientation.rotation.Y.toFixed(3) +
            " " + boneOrientation.rotation.Z.toFixed(3) +
            " " + boneOrientation.rotation.W.toFixed(3));
});
    
```

L'animation du personnage provenant du script JavaScript permettant de modifier la scène virtuelle :

```

var jsonMembers = {};
var old = {};
// json with all the related members for the animation
var relatedMembers = [
    { "id" : "SCSL" , "A" : "shouldercenter" , "B" : "shoulderleft" },
    { "id" : "SEL" , "A" : "shoulderleft" , "B" : "elbowleft" },
    { "id" : "EWL" , "A" : "elbowleft" , "B" : "wristleft" },
    { "id" : "WHL" , "A" : "wristleft" , "B" : "handleft" },
    { "id" : "HESC" , "A" : "head" , "B" : "shouldercenter" },
    { "id" : "SCSR" , "A" : "shouldercenter" , "B" : "shoulderright" },
    { "id" : "SER" , "A" : "shoulderright" , "B" : "elbowright" },
    { "id" : "EWR" , "A" : "elbowright" , "B" : "wristright" },
    { "id" : "WHR" , "A" : "wristright" , "B" : "handright" },
    { "id" : "SCS" , "A" : "shouldercenter" , "B" : "spine" },
    { "id" : "SHC" , "A" : "spine" , "B" : "hipcenter" },
    { "id" : "HCHR" , "A" : "hipcenter" , "B" : "hipright" },
    { "id" : "HKR" , "A" : "hipright" , "B" : "kneeright" },
    { "id" : "KAR" , "A" : "kneeright" , "B" : "ankleright" },
    { "id" : "AFR" , "A" : "ankleright" , "B" : "footright" },
    { "id" : "HCHL" , "A" : "hipcenter" , "B" : "hipleft" },
    
```

```

{
    "id" : "HKL" , "A" : "hipleft" , "B" : "kneyleft" ,
    "id" : "KAL" , "A" : "kneyleft" , "B" : "ankleleft" ,
    "id" : "AFL" , "A" : "ankleleft" , "B" : "footleft" } ] ;

// animate and move the skeleton
function animate(moyenne){

    // create the skeleton and set the data received of the kinect
    jQuery.each(position, function() {
        var name = this.name;

        $("#" + name).attr("translation",
            position[name].x +
            " " + position[name].y +
            " " + position[name].z);
    });

    // relate the members
    for (var k=0; k<relatedMembers.length; k++)
        buildSkeleton(relatedMembers[k]);

    move();
}

// creation of all the skeleton
function buildSkeleton(members){
    if ((position[members.A] != undefined)&(position[members.B] != undefined)) {

        // -----shoulder - elbow -----
        old[members.id] = {"distance" : null, "angle" : null}
        jsonMembers[members.id] = {
            "distance": calcDistance(position[members.A], position[members.B]),
            "angle": calcAngle(position[members.A], position[members.B])
        }

        positionning(members.id, position[members.A]);
        old[members.id].distance = scaling(members.id,
            jsonMembers[members.id].distance,
            old[members.id].distance);
        old[members.id].angle = anime(members.id,
            jsonMembers[members.id].angle,
            old[members.id].angle);
    }
}

// find the distance between 2 points on a cartesien plan
function calcDistance(p1, p2){
    return Math.sqrt(Math.pow((p2.x - p1.x),2) + Math.pow((p2.y - p1.y),2));
}

// find the angle between 2 points with te arctan2
// ref : http://www.w3schools.com/jsref/jsref_atan2.asp
function calcAngle(A, B){
    return -Math.atan2(B.y - A.y, A.x - B.x);
}

// put the member in his right place in the skeleton
function positionning(id, pos){
    $("#" + id).attr("translation", (pos.x) + " " + (pos.y - 0.5) + " " + pos.z);
}

// adapt the size of the membre dans rotate in fonction of the kinect

```

```
function scaling(id, distance, old){  
    var distance = distance.toFixed(2);  
  
    if (distance != old){  
        $("#" + id).attr("scale", distance + " " + DIV_DISTANCE + " " + DIV_DISTANCE);  
    }  
    old = distance;  
}  
  
return old;  
}  
  
// animate the members of the character  
function anime(id, angle, old){  
    var angle = angle.toFixed(2);  
    if (angle != old){  
        $("#" + id).attr("rotation", "0 0 1 " + angle);  
        old = angle;  
    }  
    return old;  
}
```

Annexe E

Illustrations différence entre X3DOM et X3D

Comme mentionné dans le chapitre 3.3, *X3DOM* et X3D sont semblables.
Voici l'exemple de code *X3DOM* :

```
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <title>My first X3DOM page</title>
    <script src='http://www.x3dom.org/download/x3dom.js'></script>
    <link rel='stylesheet' type='text/css'
          href='http://www.x3dom.org/download/x3dom.css' />
  </head>
  <body>
    <x3d width='500px' height='400px'>
      <scene>
        <shape>
          <appearance>
            <material diffuseColor='1 0 0'></material>
          </appearance>
          <box></box>
        </shape>
        <transform translation=' -3 0 0'>
          <shape>
            <appearance>
              <material diffuseColor='0 1 0'></material>
            </appearance>
            <cone></cone>
          </shape>
        </transform>
        <transform translation=' 3 0 0'>
          <shape>
            <appearance>
              <material diffuseColor='0 0 1'></material>
            </appearance>
            <sphere></sphere>
          </shape>
        </transform>
      </scene>
    </x3d>
  </body>
</html>
```

Voici l'exemple d'un cylindre créé sur *Blender* puis exporté au format X3D :

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.web3d.org/
  specifications/x3d-3.0.dtd">
<X3D version="3.0" profile="Immersive" xmlns:xsd="http://www.w3.org/2001/XMLSchema-
  instance" xsd:noNamespaceSchemaLocation="http://www.web3d.org/specifications/
  x3d-3.0.xsd">
  <head>
    <meta name="filename" content="test.x3d" />
    <meta name="generator" content="Blender 2.72 (sub 0)" />
  </head>
  <Scene>
    <NavigationInfo headlight="true"
      visibilityLimit="0.0"
      type='EXAMINE', ANY',
      avatarSize="0.25, 1.75, 0.75"
    />
    <Background DEF="WO_World"
      groundColor="0.051 0.051 0.051"
      skyColor="0.051 0.051 0.051"
    />
    <Transform DEF="ShapeIndexedFaceSet_TRANSFORM"
      translation="-0.501223 0.476330 -0.494918"
      scale="0.488878 0.488878 0.488878"
      rotation="0.571855 0.580078 0.580078 2.102657"
    >
    <Transform DEF="ShapeIndexedFaceSet_ifs_TRANSFORM"
      translation="-0.000000 0.000000 -0.000000"
      scale="1.000000 1.000000 1.000000"
      rotation="-0.000000 -0.000000 -0.000000 0.000000"
    >
    <Group DEF="group_ME_ShapeIndexedFaceSet">
      <Shape>
        <IndexedFaceSet solid="true"
          coordIndex="1 2 4 3 -1 3 4 6 5 -1 5 6 8 7
          -1 7 8 10 9 -1 9 10 12 11 -1 11 12 14 13 -1 13 14 16 15 -1 15 16 18 17 -1 17
          18 20 19 -1 19 20 22 21 -1 21 22 24 23 -1 23 24 26 25 -1 25 26 28 27 -1 27 28
          30 29 -1 29 30 32 31 -1 31 32 34 33 -1 33 34 36 35 -1 35 36 38 37 -1 37 38 40
          39 -1 39 40 42 41 -1 41 42 44 43 -1 43 44 46 45 -1 45 46 48 47 -1 47 48 50 49
          -1 49 50 52 51 -1 51 52 54 53 -1 53 54 56 55 -1 55 56 58 57 -1 57 58 60 59 -1
          59 60 62 61 -1 4 8 6 -1 12 10 8 -1 16 14 12 -1 20 18 16 -1 24 22 20 -1 28 26
          24 -1 32 30 28 -1 36 34 32 -1 40 38 36 -1 44 42 40 -1 48 46 44 -1 52 50 48 -1
          56 54 52 -1 60 58 56 -1 64 62 60 -1 4 2 64 -1 4 12 8 -1 20 16 12 -1 28 24 20
          -1 36 32 28 -1 44 40 36 -1 52 48 44 -1 60 56 52 -1 4 64 60 -1 4 20 12 -1 36 28
          20 -1 52 44 36 -1 4 60 52 -1 4 36 20 -1 4 52 36 -1 2 1 63 64 -1 61 62 64 63
          -1 1 61 63 -1 57 59 61 -1 53 55 57 -1 49 51 53 -1 45 47 49 -1 41 43 45 -1 37
          39 41 -1 33 35 37 -1 29 31 33 -1 25 27 29 -1 21 23 25 -1 17 19 21 -1 13 15 17
          -1 9 11 13 -1 5 7 9 -1 1 3 5 -1 1 57 61 -1 49 53 57 -1 41 45 49 -1 33 37 41 -1
          25 29 33 -1 17 21 25 -1 9 13 17 -1 1 5 9 -1 1 49 57 -1 33 41 49 -1 17 25 33
          -1 1 9 17 -1 1 33 49 -1 1 17 33 -1 "
        >
        <Coordinate DEF="coords_ME_ShapeIndexedFaceSet"
          point="0.000000 0.000000 0.000000 0.000000
          1.000000 -1.000000 0.000000 1.000000 1.000000 0.195090 0.980785 -1.000000
          0.195090 0.980785 1.000000 0.382683 0.923880 -1.000000 0.382683 0.923880
          1.000000 0.555570 0.831470 -1.000000 0.555570 0.831470 1.000000 0.707107
          0.707107 -1.000000 0.707107 0.707107 1.000000 0.831470 0.555570 -1.000000
          0.831470 0.555570 1.000000 0.923880 0.382683 -1.000000 0.923880 0.382683
          1.000000 0.980785 0.195090 -1.000000 0.980785 0.195090 1.000000 1.000000
          0.000000 -1.000000 1.000000 0.000000 1.000000 0.980785 -0.195090 -1.000000
          0.980785 -0.195090 1.000000 0.923880 -0.382683 -1.000000 0.923880 -0.382683
        >
      </Shape>
    </Group>
  </Transform>
</Scene>
</X3D>
```

```

1.000000  0.831470  -0.555570  -1.000000  0.831470  -0.555570  1.000000  0.707107
-0.707107  -1.000000  0.707107  -0.707107  1.000000  0.555570  -0.831470  -1.000000
0.555570  -0.831470  1.000000  0.382683  -0.923880  -1.000000  0.382683  -0.923880
1.000000  0.195090  -0.980785  -1.000000  0.195090  -0.980785  1.000000  -0.000000
-1.000000  -1.000000  -0.000000  -1.000000  1.000000  -0.195091  -0.980785  -1.000000
-0.195091  -0.980785  1.000000  -0.382684  -0.923879  -1.000000  -0.382684
-0.923879  1.000000  -0.555571  -0.831469  -1.000000  -0.555571  -0.831469  1.000000
-0.707107  -0.707106  -1.000000  -0.707107  -0.707106  1.000000  -0.831470  -0.555570
-1.000000  -0.831470  -0.555570  1.000000  -0.923880  -0.382683  -1.000000
-0.923880  -0.382683  1.000000  -0.980785  -0.195089  -1.000000  -0.980785  -0.195089
1.000000  -1.000000  0.000001  -1.000000  -1.000000  0.000001  1.000000  -0.980785
0.195091  -1.000000  -0.980785  0.195091  1.000000  -0.923879  0.382684  -1.000000
-0.923879  0.382684  1.000000  -0.831469  0.555571  -1.000000  -0.831469  0.555571
1.000000  -0.707106  0.707108  -1.000000  -0.707106  0.707108  1.000000  -0.555569
0.831470  -1.000000  -0.555569  0.831470  1.000000  -0.382682  0.923880  -1.000000
-0.382682  0.923880  1.000000  -0.195089  0.980786  -1.000000  -0.195089  0.980786
1.000000  "
    />
</IndexedFaceSet>
</Shape>
</Group>
</Transform>
</Transform>
</Scene>
</X3D>

```

Annexe F

Fichiers X3DOM

Le fichier x3dom.js étant illisible il ne vous sera pas transmis cependant voici le fichier x3dom.css permettant d'afficher de la réalité virtuelle sur le Web :

```
/*
 * X3DOM JavaScript Library
 * http://www.x3dom.org
 *
 * (C) 2009 Fraunhofer IGD, Darmstadt, Germany
 * Dual licensed under the MIT and GPL
 *
 * Based on code originally provided by
 * Philip Taylor: http://philip.html5.org
 */

body {
    background-color: white;
    font-family: Helvetica, sans-serif;
    font-size: 12px;
}

X3D, x3d {
    position: relative; /* in order to be able to position stat-div within X3D */
    float: left; /* float the element so it has the same size like the
                  canvas */
    cursor: pointer;
    margin: 0;
    padding: 0;
    border: 1px solid #000;
}

object {
    margin: 0;
    padding: 0;
    border: none;
    z-index: 0;
    width: 100%;
    height: 100%;
    float: left;
}

X3D:hover,
x3d:hover,
.x3dom-canvas:hover {
```

```
    -webkit-user-select: none;
    -webkit-touch-callout: none;
}

.x3dom-canvas {
    border:none;
    cursor:pointer;
    cursor:-webkit-grab;
    cursor:grab;
    width:100%;
    height:100%;
    float:left;
}

.x3dom-canvas-mousedown {
    cursor:-webkit-grabbing;
    cursor:grabbing;
}

.x3dom-canvas:focus {
    outline:none;
}

.x3dom-progress {
    margin: 0;
    padding: 6px 8px 0px 26px;
    left: 0px;
    top: 0px;
    position: absolute;
    color: #0f0;
    font-family: Helvetica, sans-serif;
    line-height:10px;
    font-size: 10px;
    min-width: 45px;
    min-height: 20px;
    border: 0px;
    background-position: 4px 4px;
    background-repeat: no-repeat;
    background-color: #333;
    background-color: rgba(51, 51, 51, 0.9);
    z-index: 100;
    background-image: url('data:image/gif;[...]');
}

.x3dom-progress .bar span {
    position: absolute;
    left: 0;
    top: 0;
    line-height: 20px;
    background-color: red;
}

.x3dom-statdiv {
    margin: 0;
    padding: 0;
    right: 10px;
    top: 10px;
    position: absolute;
    color: #0f0;
    font-family: Helvetica, sans-serif;
    line-height:10px;
    font-size: 10px;
```

```
width: 75px;
height: 70px;
border: 0px;
}

#x3dom-state-canvas {
    margin: 2px;
    padding: 0;
    right: 0%;
    top: 0%;
    position: absolute;
}

#x3dom-state-viewer {
    position: absolute;
    margin: 2px;
    padding: 5px;
    width: 135px;
    top: 0%;
    right: 0%;
    opacity: 0.9;
    background-color: #323232;
    z-index: 1000;
    font-family: Arial, sans-serif;
    color: #C8C8C8;
    font-weight: bold;
    text-transform: uppercase;
    cursor: help;
}

.x3dom-states-head {
    display: block;
    font-size: 26px;
}

.x3dom-states-head2 {
    font-size: 10px;
}

.x3dom-states-list {
    float: left;
    width: 100%;
    border-top: 1px solid #C8C8C8;
    list-style: none;
    font-size: 9px;
    line-height: 16px;
    margin: 0;
    padding: 0;
    padding-top: 2px;
}

.x3dom-states-item {
    width: 100%;
    float: left;
}

.x3dom-states-item-title {
    float: left;
    margin-left: 2px;
}

.x3dom-states-item-value {
```

```
    float: right;
    margin-right: 2px;
}

.x3dom-touch-marker {
    display: inline;
    padding: 5px;
    border-radius: 10px;
    position: absolute;
    font-family: Helvetica, sans-serif;
    line-height:10px;
    font-size: 10px;
    color: darkorange;
    background: cornsilk;
    opacity: 0.6;
    border: 2px solid orange;
    z-index: 200;
}

.x3dom-logContainer {
    border: 2px solid olivedrab;
    height: 200px;
    padding: 4px;
    overflow: auto;
    white-space: pre-wrap;
    font-family: sans-serif;
    font-size: x-small;
    color: #00ff00;
    background-color: black;
    margin-right: 10px;
}

.x3dom-nox3d {
    font-family: Helvetica, sans-serif;
    font-size: 14px;
    background-color: #eb7a7a;
    padding: 1em;
    opacity: 0.75;
}

.x3dom-nox3d p {
    color: #fff;
    font-size: 14px;
}

.x3dom-nox3d a {
    color: #fff;
    font-size: 14px;
}

/* self-clearing floats */
.group:after {
    content: " ";
    display: block;
    height: 0;
    clear: both;
    visibility: hidden;
}
```

Annexe G

Tutoriel d'installation des outils et utilisation du projet Virtual-Vertigo

Afin de vous permettre d'installer et d'utiliser le projet *Virtual-Vertigo*, ci-dessous, un tutoriel d'installation et manuel utilisateur.

Tutoriel Virtual-Vertigo

1 CONSTRUCTION DES GOOGLE CARDBOARD

Pour construire des *Google CardBoard* il faut :

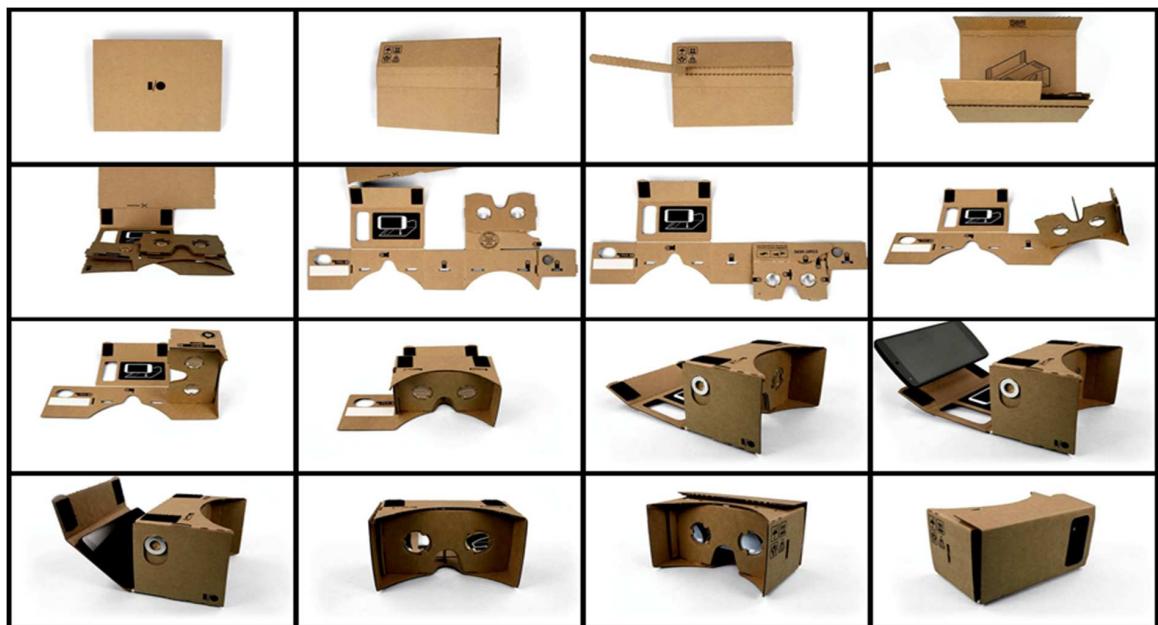
- Du carton pas trop épais (1-2mm)
- Des lentilles biconvexes
- Un tag NFC
- Un aimant en céramique
- Un aimant en néodyme
- Du velcro pour tenir le smartphone



Le matériel de construction se trouve facilement sur *Amazon*. Lien¹ pour l'achat d'un kit complet.



Les patrons de conception et instructions de montage sont fournis par Google sur leur page². Il suffit simplement de télécharger les documents, les imprimer, les coller sur le carton, découper et assembler pour obtenir les *Google CardBoard*. Une vidéo³ expliquant les étapes pour construire les *Google CardBoard*.



¹ Matériel pour la construction : http://www.amazon.com/AM-CARDBOARD%C2%AE-Complete-Cardboard-Project/dp/BooLM36DUK/ref=pd_sim_cps_14?ie=UTF8&refRID=1SNC11VTMPQND1MF5ZTE

² Page des Google CardBoard : <https://www.google.com/get/cardboard/get-cardboard.html>

³ Construction : <https://www.youtube.com/watch?v=3YopUPZErl>

2 INSTALLATION ET EXEMPLE D'UTILISATION DE NODEJS

Allez sur le site : <http://nodejs.org/download/>

Infos : Lors de son installation, NodeJS installe également npm qui nous permet de télécharger et installer des modules à NodeJS.

Windows

Téléchargez le fichier **msi** pour Windows 32 ou 64 bits disponible sur le site internet et exécutez-le. Laissez les paramètres par défaut.

Linux

Via terminal

Ouvrez un terminal et tapez.

```
sudo apt-get install nodejs npm  
sudo apt-get update
```

Via internet

Téléchargez le fichier .tar.gz pour linux 32 ou 64 bits et exécutez-le. Laissez les paramètres par défaut.

Utilisation de *NodeJS*

NodeJS est maintenant prêt à être utilisé. Voici un exemple de serveur web afin que vous puissiez tester :

```
// Load the http module to create an http server.  
var http = require('http');  
  
// Configure our HTTP server to respond with Hello World to all requests.  
var server = http.createServer(function (request, response) {  
    response.writeHead(200, {"Content-Type": "text/plain"});  
    response.end("Hello World\n");  
});  
  
// Listen on port 8000, IP defaults to 127.0.0.1  
server.listen(8000);  
  
// Put a friendly message on the terminal  
console.log("Server running at http://127.0.0.1:8000/");
```

Il suffit de lancer un terminal, de vous placer dans le répertoire du serveur et de taper.
Infos : le .js n'a pas besoin d'être spécifié.

```
Node <nomserver.js>
```

Après avoir tapez cette ligne de commande, ouvrez un navigateur Web et tapez afin d'afficher **Hello World**

```
Localhost :8000
```

Npm

C'est un package manager permettant d'installer des packages sur *NodeJS* permettant de compléter NodeJS en fonction de ce que vous voulez faire avec.

Vous pouvez chercher des packages sur <https://www.npmjs.com/#explicit>.

Voici les packages les plus utilisés :

- Socket.io qui permet d'utiliser les WebSockets et de facilement les manipuler
- ExpressJS qui permet de créer et gérer une application web plus facilement
- AngularJS qui permet d'étendre le langage HTML par de nouvelles balises et attributs
- MongoDB qui permet de gérer des bases de données

Pour installer un module, ouvrez un terminal, placez-vous dans le dossier où se trouve votre serveur node et tapez.

```
npm install <nomdu module>
```

3 INSTALLATION ET TEST DU KINECT V1

Attention : Vous devez être sur Windows 7 ou plus pour pouvoir utiliser le Kinect v1

SDK

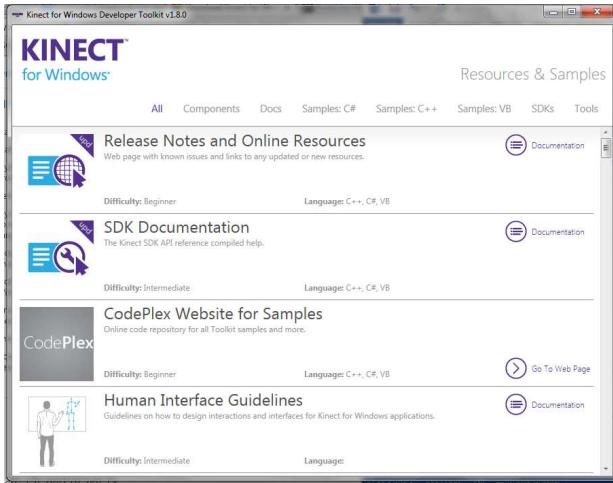
Allez sur <http://www.microsoft.com/en-us/download/details.aspx?id=40278> et téléchargez le SDK 1.8. Exécutez KinectSDK-v1.8-Setup.exe et laissez tous les paramètres par défaut.

Developer Toolkit

Allez sur <http://go.microsoft.com/fwlink/?LinkID=323589> et téléchargez le *Toolkit*. Exécutez KinectDeveloperToolkit-v1.8.0-Setup.exe et laissez tous les paramètres par défaut.

Test du Kinect

Branchez votre Kinect sur votre ordinateur et ouvrez le *Toolkit*.



Choisissez un exemple sous *Sample C# ou Sample C++ ou Sample VB* et cliquez sur play.

4 UTILISATION DE L'APPLICATION VIRTUAL-VERTIGO

4.1 RÉCUPÉRATION DES SOURCES DU PROJET

Récupérez le zip contenant l'application *Virtual-Vertigo* se trouvant sur le CD annexé au rapport. Dézippez-le.

4.2 LANCEMENT DU SERVEUR

Connectez le Kinect v1 à votre ordinateur et placez la planche sur le sol face au Kinect.

Puis allez dans le dossier *Application* de l'archive dézippée auparavant et lancez le fichier *launcher.bat*.

Infos : Ce fichier bat va lancer l'application du Kinect et le serveur Web NodeJS Virtual-Vertigo.

Lorsque le *serveur Web Virtual-Vertigo* est lancé, il fournit l'adresse ip et le port du PC. Veuillez **retenir** ou noter **cette adresse ip et port**.

4.3 LANCEMENT DE LA SIMULATION

Sur le PC, ouvrez un navigateur et allez sur <*adresse ip :port/index.html ?mode=2*> (**adresse ip et port noté auparavant**) .

Sur votre smartphone, ouvrez un navigateur et allez sur <*adresse ip :port/index.html ?mode=1*> (**adresse ip et port noté auparavant**) et cliquez sur **fullscreen**

Placez votre smartphone dans les Google Cardboard de cette manière :



Pour finir, placez-vous à l'extrémité de la planche et accrochez les *Google CardBoard* sur votre tête en utilisant le velcro.

Profitez de votre simulation !!

Bibliographie

- [1] Wikipédia. Page wikipédia sur webhook. <https://en.wikipedia.org/wiki/Webhook>.
- [2] Wikipédia. Page wikipédia sur les serious games. https://en.wikipedia.org/wiki/Serious_game.
- [3] LudoMedic. Plateforme médicale vidéoludique (serious game). <http://www.ludomedic.com/>.
- [4] CCCP. Cccp, développeur de serious games et jeux vidéo. <http://cccp.fr/>.
- [5] Wikipédia. Page wikipédia sur l'acrophobie (la peur du vide). <https://fr.wikipedia.org/wiki/Acrophobie>.
- [6] Léa Galanopoulo. Article "guérir le vertige grâce à la réalité virtuelle". *CNRS Le journal*, Août 2014. <https://lejournal.cnrs.fr/articles/guerir-le-vertige-grace-a-la-realite-virtuelle>.
- [7] Daniel MESTRE. Projet "contrôle du stress par cyberthérapie", Mars 2015. http://www.cnrs.fr/mi/IMG/pdf/itmm_pres17marsmestre.pdf.
- [8] Google. Page du site des google cardboard pour le montage. <https://www.google.com/get/cardboard/get-cardboard/>.
- [9] Wikiédia. Description de la stéréoscopie. <https://fr.wikipedia.org/wiki/Stéréoscopie>
- [10] Google. Application cardboard.
- [11] Google. Application spotlight stories. <https://play.google.com/store/apps/details?id=com.google.android.spotlightstories>.
- [12] Microsoft. Page web contenant les pré-requis pour le kinect v1. <https://msdn.microsoft.com/en-us/library/hh855359.aspx>.
- [13] Microsoft. Description de l'orientation des points du squelette. <https://msdn.microsoft.com/en-us/library/hh973073.aspx>.
- [14] Wikipédia. Description des quaternions pour la rotation dans l'espace. https://fr.wikipedia.org/wiki/Quaternions_et_rotation_dans_l'espace.
- [15] Wikipédia. Description des matrices de rotations. https://fr.wikipedia.org/wiki/Matrices_de_rotation.
- [16] Microsoft. Page web contenant des informations sur chaque version du sdk. <https://msdn.microsoft.com/en-us/library/jj663803.aspx>.
- [17] joeandaverde. Dépôt git de joeandaverde sur lequel se trouve socket.io-csharp-client, 2012. <https://github.com/joeandaverde/socket.io-csharp-client>.
- [18] kerryjiang. Dépôt git de kerryjiang sur lequel se trouve websocket4net. <https://github.com/kerryjiang/WebSocket4Net>.
- [19] Communauté X3DOM. Page web de x3dom. <http://www.x3dom.org/>.

BIBLIOGRAPHIE

- [20] Wikipédia. Page wikipédia sur dom (document objet model). https://fr.wikipedia.org/wiki/Document_Object_Model.
- [21] Wikipédia. Page wikipédia sur x3d (extensible 3d). https://fr.wikipedia.org/wiki/Extensible_3D.
- [22] Wikipédia. Page wikipédia sur les websockets. <https://fr.wikipedia.org/wiki/WebSocket>.
- [23] Communauté MeteorJS. Page web de meteorjs. <https://www.meteor.com/>.
- [24] Communauté NodeJS. Page web de nodejs. <https://nodejs.org/>.
- [25] Socket.io. Site internet de socket.io, 2012. <http://socket.io/>.
- [26] json. Site internet de json. <http://json.org>.
- [27] X3DOM. Liste de tout les nodes possibles sur x3dom. <http://doc.x3dom.org/author/nodes.html>.
- [28] Wikipédia. Page wikipédia expliquant atan2. <https://fr.wikipedia.org/wiki/Atan2>.
- [29] nguyer. Dépôt git de nguyer sur lequel se trouve node_kinect. <https://github.com/nguyer/node-kinect>.
- [30] GNU Operating System. Librairie c permettant de communiquer avec les périphériques usb. <http://www.libusb.org/>.
- [31] OpenKinect. Dépôt git de openkinect sur lequel se trouve libfreenect. <https://github.com/OpenKinect/libfreenect>.
- [32] OpenKinect. Site de openkinect. <http://openkinect.org/wiki>.
- [33] Vangos Pterneas. Dépôt git de vangos pterneas contenant kinecthtml5. <https://github.com/Vangos/KinectHtml5>.