CrowdSimulator

Clément JANISSET

Laboratoire de psychophysique
et de perception visuelle de
Montréal

I.U.T. Clermont-Ferrand
Département Imagerie Numérique

Année Universitaire 2013-2014
31 Mars 2014 - 20 Juin 2014

Stage réalisé par Clément JANISSET

Au Laboratoire de psychophysique

et de perception visuelle de Montréal

Sur la réalisation d'un simulateur de foule (CrowdSimulator)

Du 31 Mars 2014 au 20 Juin 2014

I.U.T. Clermont-Ferrand 1, Département Imagerie Numérique

Année universitaire 2013-2014

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui m'ont accompagné tout au long de mon stage et qui ont ainsi participé à sa réussite.

Je voudrais tout particulièrement remercier Monsieur **FAUBERT Jocelyn** ainsi que toute son équipe pour m'avoir accueilli au sein de son laboratoire et proposé un stage très intéressant, innovant et unique. Mais également pour avoir mis à notre disposition tous les moyens nécessaires à la réalisation et l'aboutissement du projet.

Egalement un énorme merci à Monsieur **SUTYUSHEV Vadim** pour son immense patience, sa gentillesse et sa disponibilité malgré les (trop) nombreuses fois où le CAVE s'est arrêté de fonctionner à cause du simulateur.

Monsieur **BISCHOFF Pierre Yves**, Madame **RIVET Marie-Laure** et Monsieur **SAUVAGE Vincent** pour m'avoir trouvé ce stage et guidé dans les démarches administratives tout au long de ce dernier.

Ainsi que l'équipe qui était avec moi durant le projet (**THOUVENET Benoit**, **GAY Kevin** et **SCHIAVI Barbara**).

RESUME

J'ai du concevoir en collaboration avec une équipe de 3 personnes un simulateur de foule à destination d'un CAVE.

Un CAVE est un appareil de réalité augmenté de la grandeur d'une pièce utilisant 4 écrans 3D (un en face, un à droite, un à gauche et un au sol par rapport à l'utilisateur). Cet appareil permet à une ou plusieurs personnes d'être immergées dans un monde virtuel.

Ce simulateur sera destiné à tester et à affiner les capacités cérébrales des sujets qui l'utiliseront. En effet, une fois le simulateur de foule lancé sur le CAVE, les sujets auront à suivre du regard un ou plusieurs éléments. A titre d'exemple, cela leur permettra de traiter plus d'informations, de mieux exploiter leur vision périphérique ou de suivre de plus en plus de cibles simultanément au fur et à mesure des tests.

SOMMAIRE

INTRODUCTION

PRESENTATION DU PROJET

During my internship, I had to build a crowd simulator for the CAVE with a team composed of three other students. A CAVE is a device that can bring you in a virtual world with four 3D screens (one in front of you, two on your sides and a last on the ground). To build the program, we use the software Unity that allow us to create a program quickly and easily adaptable for the CAVE.

This simulator is based on the idea of the Neurotracker software. The aim of Neurotracker is to increase people's abilities who use the software. Indeed, it ask them to track between one or four balls out of eight on the screen, mix up the balls, stop them, and after ask people which balls they must track and check if they success to track the balls.

So the crowd simulator ask people to track some virtual people in the crowd, run the simulation, then stop, and ask them to find out the correct virtual people. The added value of the crowd simulator compared to Neurotracker is that add something we call the noise. This noise can be create by the cloths, the environment, some objects... All things that's not in Neurotracker and can disturb people during the test.

This crowd simulation will be used later by researcher to study patients' reactions facing the crowd, their abilities to find some specifics persons, increase their focus, their resistance to fatigue, their peripheral awareness, and much more ! It can also used on professional athlete to boost their gameshape like reacting faster, avoid collisions and tackles more easily or recognize more threats for example.

Nous avions quelques contraintes imposées par Monsieur FAUBERT pour le simulateur de foule.

Tout d'abord au niveau des environnements, nous devions réaliser deux environnements différents. Le premier étant un centre commercial, et le second un carrefour de rue. Pour le centre commercial, la seule contrainte était d'avoir plusieurs étages. Quand à la rue, aucune contrainte précise n'avait été spécifiée, hormis le fait qu'il s'agisse d'un carrefour Québécois, avec feu de circulation de l'autre coté du carrefour, et rue extra large. Ce que nous n'avions pas pris en compte au début ! Les contraintes de l'environnement étant donc très faibles, cela a permis à nos modélisateurs d'exprimer leur talent et de laisser libre cours à leur imagination.

Ensuite, au niveau de la foule, nous avions deux contraintes principales. Pour la première contrainte, Monsieur FAUBERT portait tout particulièrement notre attention au réalisme de la foule. Il souhaitait une foule bien réalisée en terme de modélisation, mais aussi en terme de comportement. Pour la seconde contrainte, il fallait que la simulation soit paramétrable. C'est à dire que certains paramètres de la simulation devaient être modifiable pendant que le simulateur tournait.

Parmi ces paramètres, nous devions pouvoir gérer la vitesse des personnages, de façon à ce qu'ils aillent tous à une même allure, ou bien chacun à la leur (vitesse aléatoire).

Nous devions également faire en sorte que la foule soit cohérente ou non, c'est à dire qu'un certain pourcentage de personnage devait emprunter un chemin aléatoire, tandis que les autres suivaient un chemin prédéfinit.

Quelques réglages de temps nous étaient aussi demandés comme le temps d'initialisation (temps servant aux sujets à repérer les cibles à suivre), ou encore la durée de la simulation, durant laquelle les sujets doivent suivre les cibles du regard et ne pas les perdre de vue.

Nous devions également gérer l'activité du sujet par rapport à cette foule, c'est à dire que le sujet pouvait prendre part à la scène, être actif (se déplacer et s'approcher des personnes constituant la foule) ou bien être passif (un simple spectateur avec la foule au loin).

Et enfin, nous devions gérer le déroulement de la simulation, c'est à dire faire en sorte que l'expérience se déroule facilement et sans encombre en respectant les étapes suivantes:

- Initialisation de la foule avec le temps arrêté et repérage des cibles.
- Exécution de la simulation sans aucun indice sur les cibles.
- Arrêt du temps et indexation des personnages afin que les sujets puissent identifier les personnages facilement.
- Affichage des cibles en surbrillance afin de vérifier les résultats donnés par le sujet.

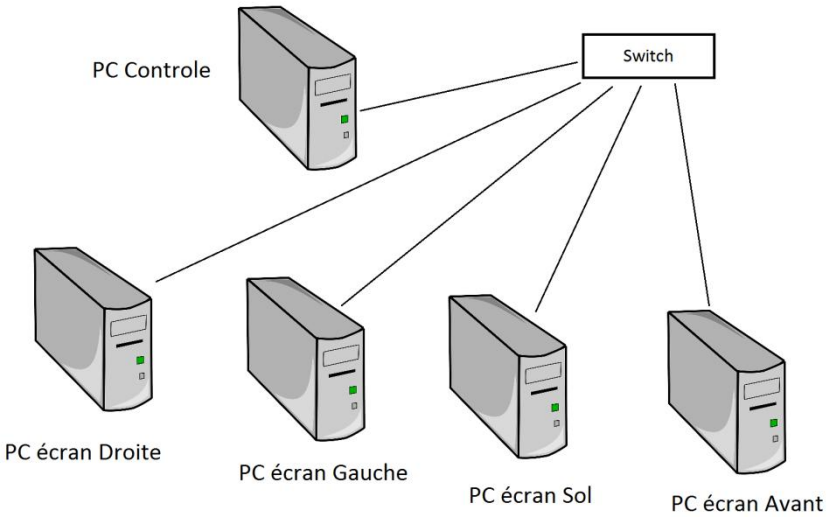
PRESENTATION DU CAVE

Comme mentionné précédemment, le CAVE est composé de 4 écrans 3D afin de créer une immersion totale dans un monde virtuel. Afin de fonctionner, le CAVE est composé de 5 ordinateurs: un pour chaque écran, et un dernier servant d'ordinateur de contrôle. Ces 5 ordinateurs forment ce que l'on appelle des clusters[1]. Ils sont reliés ensemble grâce à un switch[2] à la vitesse d'1Gb/s.

Le fait que chaque écran possède son propre ordinateur nous permet d'avoir beaucoup plus de puissance graphique car l'image à calculer restera relativement petite. Cela se traduit par la possibilité d'afficher plus de modèles et beaucoup plus de détails à l'écran (pour le plus grand bonheur des modélisateurs). Mais cet avantage se retourne très vite contre les programmeurs.

En effet, cela signifie, dans notre cas, que chaque ordinateur devra exécuter la même simulation en même temps, ce qui nous impose une énorme contrainte au niveau de la foule : faire en sorte que la foule soit exactement la même sur 4 ordinateurs en même temps. Mais également que celle-ci soit aléatoire afin d'augmenter le réalisme et éviter un phénomène d'accoutumance des sujets (la simulation n'a plus aucun intérêt si les sujets connaissent par cœur le trajet de chaque personnage). Et c'est là tout l'enjeu du CAVE, parvenir à un résultat identique en temps réel sur 4 ordinateurs en même temps.

Face à ces contraintes, nous avons dû adapter notre programmation et trouver des solutions pour parvenir à un résultat.



(Schéma de la structure du CAVE)

[1] Cluster: système informatique composé d'unités de calcul (micro-processeurs, cœurs, unités centrales) autonomes qui sont reliées entre elles à l'aide d'un réseau de communication.
[2] Switch: équipement ou appareil qui permet l'interconnexion d'appareils communicants comme des ordinateurs.

DEVELOPPEMENT

ORGANISATION

Après nous avoir fait visiter le laboratoire et expliqué le cahier des charges, Monsieur FAUBERT nous a laissé nous concerter afin de définir les bases du simulateur de foule. Il attendait de nous un travail autonome et nous accordait une confiance totale afin de mener à terme ce projet.

Nous avons donc dû nous organiser afin de pouvoir tous travailler en continu sur le même projet sans se gêner. Pour répertorier tous les avancements du projet en ligne, nous avons utilisé un GIT[3]. La première étape a donc été de choisir un GIT qui nous permettait de travailler sans que des utilisateurs externes puissent consulter le projet. Nous avons donc choisi BitBucket qui nous permet de créer des dépôts de projets privés sans avoir besoin de payer un abonnement (contrairement à GitHub).

Coté programmation, nous avons séparé le travail en tâches à réaliser par ordre de priorité. Chaque personne choisissait une tâche à la fois, et passait à une autre tâche lorsqu'elle en avait fini avec la précédente.

Pour les modélisateurs, une personne devait se charger de la modélisation des personnages, ainsi que de leur animation; et l'autre de l'architecture et du design des deux scènes.

Afin de permettre à Monsieur FAUBERT d'avoir un premier retour du projet sans avoir à payer les licences pour les logiciels dont nous avions besoin durant le développement, nous avons décidé de réaliser le projet en utilisant le mois d'essai de ceux-ci. Ainsi, nous avons activé le mois d'essai dans un premier temps sur un ordinateur, puis sur un autre à la fin du mois, et enfin sur le CAVE le dernier mois afin de pouvoir tester notre application en condition réelle à la fin du stage.

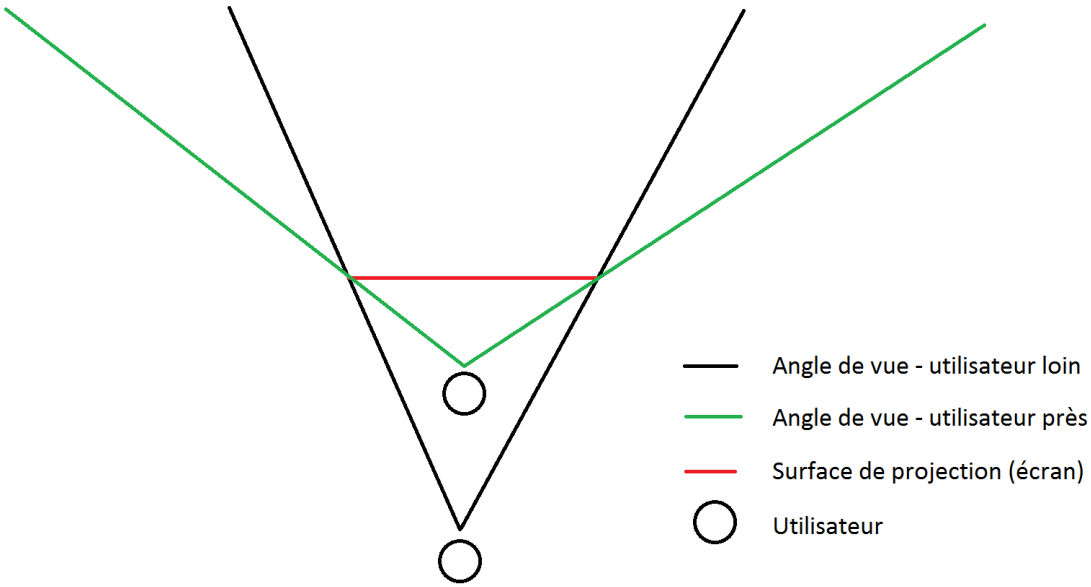
ADAPTATION D'UN PROJET UNITY SUR LE CAVE

Notre première tâche a tout d'abord consisté à réaliser un gros travail de recherche afin d'adapter une scène Unity[4] sur le CAVE. Il s'agissait de la tâche la plus essentielle au projet, et celle qui allait grandement déterminer le temps de développement du projet en fonction de la simplicité d'adaptation d'une scène ou non.

[3] GIT: Système permettant de travailler par version sur un même projet, permettant ainsi de conserver toutes les modifications apportées aux fichiers entre les versions.
[4] Unity: Unity est un logiciel 3D temps réel et multimédia ainsi qu'un moteur 3D/2D et physique utilisé pour la création de jeux en réseau, d'animation en temps réel, de contenu interactif comportant de l'audio, de la vidéo et des objets 3D/2D.

Nous avons donc passé nos trois premiers jours de stage à nous documenter sur la manière dont fonctionnait le CAVE, et sur les logiciels déjà existants sur le marché permettant d'exécuter du code sur celui-ci.

L'architecture du CAVE étant déjà expliquée au dessus, nous allons nous attarder sur les fonctions nécessaires à l'adaptation d'un projet sur un CAVE. Tout d'abord, il faut un logiciel capable de synchroniser les 4 ordinateurs du CAVE afin d'avoir un rendu cohérent sur les 4 écrans. Le logiciel recherché devra également pouvoir calculer le champ de vision du sujet en fonction de sa position dans le CAVE. En effet, plus le sujet se rapproche de l'écran, plus le volume de la scène affiché à l'écran devra être grand (voir schéma ci-dessous).



Nous avons commencé par chercher des solutions gratuites comme VRJuggler ou encore CAVELib. Le problème de ces logiciels est qu'ils ne peuvent pas réellement s'adapter sur un projet Unity sans un travail conséquent. De plus, ils ne sont plus mis à jour depuis longtemps (par exemple, la dernière mise à jour de VRJuggler remonte au 27 Novembre 2010) et ne possèdent quasiment aucun support.

Après avoir eu l'autorisation de notre maitre de stage, nous avons donc cherché parmi les logiciels payant et nous avons choisi MiddleVR, qui convient à toutes nos exigences et possède en plus une licence académique.

MiddleVR est un Plug-in pour Unity permettant de porter facilement des projets Unity entiers sur le CAVE. Pour cela, il suffit simplement d'importer le Plug-in dans Unity, et de placer le Préfab[5] "VRManager" de MiddleVR dans la scène. Ce Préfab permet au logiciel MiddleVR de faire la liaison avec sa librairie et ainsi réaliser les calculs de champs de vision, la disposition des caméras dans la scène et la synchronisation des clusters en toute transparence sans qu'on ait à s'en occuper.

[5] Préfab: Il s'agit d'un GameObject (objet sous Unity) préfabriqué avec des valeurs préconfigurées pour une utilisation simple, rapide et efficace.

Un script ("VrClusterObject") vient quand à lui compléter la synchronisation des objets, en synchronisant la position et la rotation de ceux-ci entre les clusters. En effet, il doit être placé sur les GameObjects[6] que l'on souhaite synchroniser. Il n'est pas ajouté automatiquement sur tout les objets afin d'éviter d'encombrer le réseau avec des informations inutiles (si un objet ne bouge pas, par exemple un banc, la synchronisation est inutile puisque l'information de position et de rotation sera toujours la même).

MiddleVR gère également les périphériques d'entrées et de sorties. Nous avons donc utilisé son gestionnaire afin de récupérer les entrées de la manette plutôt que le gestionnaire d'Unity pour la simple raison que la manette ne peut être branchée que sur un seul ordinateur en même temps, ce qui implique que les entrées soient synchronisées entre les clusters (gérées nativement par MiddleVR).

Pour finir, MiddleVR est fourni avec les instructions d'utilisation, mais ne possède aucune documentation, ce qui est le seul défaut que je reproche au logiciel. Ce point négatif est rattrapé par la qualité du support technique que nous avons eu l'occasion de contacter à deux reprises.

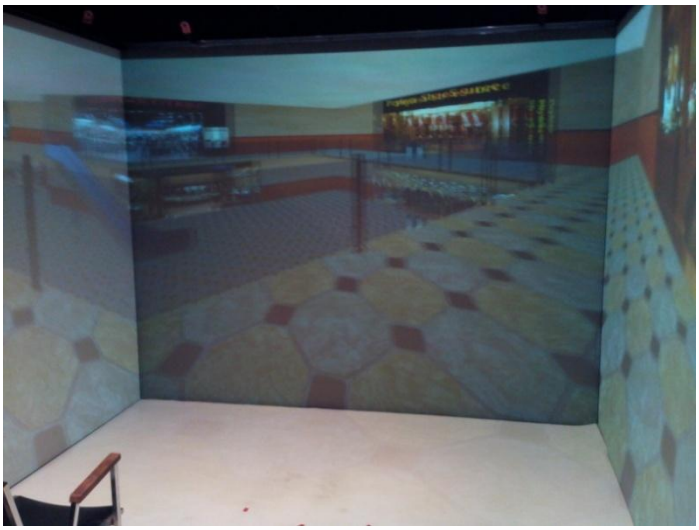
CONCEPTION DU SIMULATEUR

Le simulateur devait comporter deux scènes: un croisement de rue, et un centre commercial. Nous avons donc défini certains éléments à intégrer à ces environnements afin de les rendre plus réalistes et dynamiques.

Ainsi, pour le carrefour, nous avons décidé d'intégrer un trafic de voitures paramétrable au même titre que la foule. Des feux de circulation ont également été intégrés à la scène afin de gérer plus facilement le flux de voiture, mais aussi pour permettre de donner l'autorisation ou non à la foule de traverser. Afin de contrôler la foule et de la coordonner un minimum sur le carrefour, nous avons également intégré des passages piétons proches des feux de circulation.

Pour le centre commercial, nous avons décidé d'ajouter des escalators pour permettre à la foule, mais également à l'avatar du sujet, de changer d'étage pendant la simulation. Ce choix a été motivé par le coté dynamique de l'escalator contrairement à de simples escaliers, mais aussi en raison de l'ancien centre commercial où un escalator était déjà présent. En effet, lorsque Monsieur FAUBERT nous a présenté le CAVE pour la première fois, il nous a fait essayer les multiples applications disponibles sur celui-ci, dont l'ancienne application de centre commercial ne comportant qu'un centre commercial sans foule.

[6] GameObject: Représente une entité virtuel dans le logiciel Unity

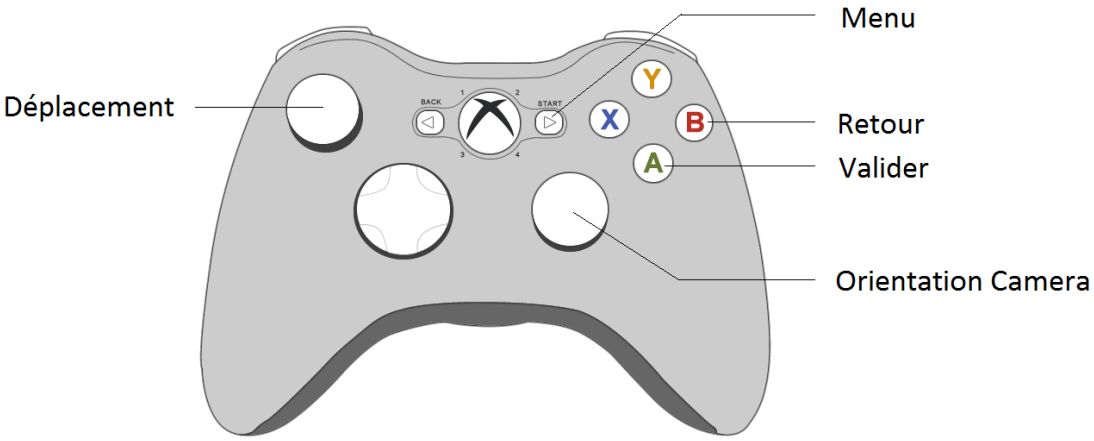


(Photo du CAVE faisant tourner l'ancienne application de centre commercial)

Nous avons également décidé de rajouter des vitrines avec lesquelles la foule devra interagir. Ainsi, lorsque qu'une personne de la foule s'approchera d'une vitrine, elle s'arrêtera devant pendant quelques secondes afin de paraitre plus réaliste.

Pour le choix de l'environnement, nous avons décidé qu'un écran de sélection devrait apparaitre au tout début de l'application afin de rediriger le sujet dans un des deux environnements. Un autre menu devrait quand à lui apparaitre à l'intérieur de la simulation afin de permettre à l'utilisateur de régler les différents paramètres de la simulation.

Pour ce qui est de l'interaction avec le CAVE, nous avons utilisé une manette de Xbox 360 sans fil comme toutes les autres applications présentes sur le CAVE. Ainsi, nous avons repris la configuration des touches standards à la plupart des applications en vue à la première personne. (Voir schéma ci-dessous)



(Configuration des touches de la simulation)

LA FOULE

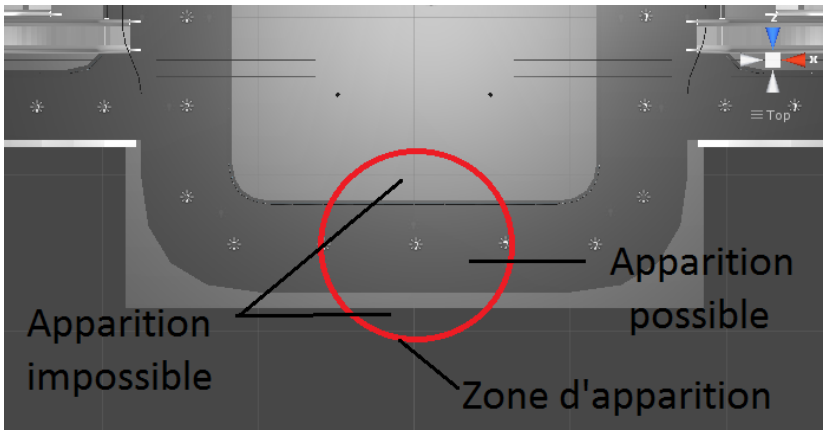
Une fois les grands axes de la simulation décidés, et en attendant les nouvelles scènes, nous avons donc démarré avec l'ancien modèle de centre commercial pour commencer nos recherches sur la foule.

LES PERSONNAGES

Afin de commencer à former une foule, nous avons dû avant tout concevoir et réaliser des personnages. Sans attendre les premiers modèles 3D de personnages, nous avons commencé avec des capsules standards présentes dans Unity. Ainsi, nous avons pu concevoir l'apparition de ces personnes sur la scène de manière aléatoire afin de former une foule directement dès le début de la simulation. Pour cela, nous avons créé deux scripts.

Dans un premier temps, nous avons le script "Spawn.cs" qui permet, une fois placé sur un GameObject, de faire apparaitre des personnages à l'emplacement de ce GameObject. Il s'agit simplement de faire apparaitre des personnages à un point donné. Le script nous sert également à informer le personnage sur l'étage sur lequel il se trouve (qui servira plus tard à l'intelligence artificielle du personnage pour le repérer dans la scène). Plus tard dans le développement du simulateur, nous avons voulu étoffer ce script en le transformant non pas en point d'apparition, mais en zone d'apparition, permettant ainsi d'obtenir une foule plus homogène dès le début de la simulation. Nous avons donc modifié ce script afin que les personnages apparaissent aléatoirement à l'intérieur d'un cercle. Cependant, ce système nous posait un léger problème : les personnages pouvaient dans certains cas apparaitre à l'extérieur de la scène ou dans le vide (par exemple les corridors très étroits du deuxième étage). Nous avons résolu ce problème simplement en testant l'existence d'un sol sous les personnages avec l'aide d'un Raycast[7] avant de les faire apparaitre. Si le test se révèle négatif, on essaye de les faire apparaitre à un autre endroit.

[7] Raycast: Fonction sous Unity permettant de lancer un rayon dans une scène sur une certaine distance et de connaitre le premier objet intercepté par ce rayon.



(Exemple d'une zone d'apparition sur la scène du centre commercial)

Ainsi, afin de pouvoir réutiliser le script simplement et rapidement, nous l'avons placé sur un GameObject vide que nous avons ensuite enregistré sous la forme d'un préfab. Pour simplifier les réglages, nous avons également décidé de rendre publique la variable correspondant au rayon de la zone d'apparition nous permettant de la régler arbitrairement dans l'éditeur Unity.

Le deuxième script gérant l'apparition de la foule est le script "SpawnManager.cs". Il permet de répartir la foule à travers les zones d'apparition. C'est lui qui va indiquer aux zones combien de personnes elles doivent faire apparaitre. SpawnManager contient aussi une liste publique de GameObject contenant les préfabs des personnages afin de pouvoir intégrer de nouveaux personnages facilement dans la foule.

Plus tard dans le développement, nous avons eu les premiers modèles de personnages avec une animation de marche. Nous avons donc rechercher comment intégrer et jouer des animations sur Unity. Unity propose un moteur d'animation extrêmement puissant et intuitif nommé "Mecanim". Mecanim nous permet de contrôler les animations grâce à de simples variables, mais aussi de gérer les changements d'animation en interpolant les points du modèle animé afin de faire en sorte que l'animation paraisse beaucoup plus fluide lors des transitions. Enfin, il permet d'appliquer la même animation sur plusieurs modèles à la seule condition que ces modèles possèdent les mêmes points d'animation.

Nous avons donc intégré quatre personnages ainsi que l'animation de marche dans le projet assez facilement et réglé une incohérence dans l'animation de marche. En effet, l'animation de marche ne bouclait pas. C'est à dire que la position de départ du personnage dans l'animation ne correspondait pas à la position de fin de l'animation.

Malheureusement nous n'avons pas pu intégrer d'autres animations, notamment des animations d'attente afin de donner plus de réalisme à la simulation. Effectivement, au terme des trois mois de stage et malgré la réalisation d'une séance de Motion

Capture[8], nous n'avons eu aucune autres animations pour la simulation. Les seules animations rendues n'étaient soit pas réaliste du tout, soit tiré d'internet et n'avaient absolument aucun rapport avec la simulation. Nous avons donc décidé de ne pas les intégrer dans le simulateur.

Enfin, les personnages devaient également être synchronisés entre les clusters, nous leur avons donc appliqué le script "VRClusterObject" fourni par MiddleVR (c.f. Chapitre "Adaptation d'un projet Unity sur le CAVE"). Ce script nous a posé un énorme problème à ce moment là. En effet, la synchronisation entre les clusters se faisant par le biais du réseau, il fallait donc à tout prix éviter de synchroniser trop d'objets. Ainsi, pour une centaine de personnages la synchronisation se déroule très bien, mais dès que l'on commence à en synchroniser plus, le réseau saccade et le nombre d'images par seconde chute rapidement jusqu'à atteindre les 1 image/seconde. Nous avons donc décidé de limiter la foule à 100 personnages maximum.

Le seul souci était lorsqu'on relançait la simulation: les personnages du test précédent continuaient à être synchronisés alors que ceux ci avaient été supprimés. Après plusieurs heures de recherche sur internet et avoir épluché les logs (historique des événements) de MiddleVR, nous nous sommes décidés à contacter le support technique de MiddleVR. Le support technique se trouvant en France, il fallait se lever le plus tôt possible en raison du décalage horaire afin de pouvoir dialoguer plus longtemps avec eux. Après avoir échangé plusieurs mails, il s'est avéré que notre souci était dû au fait qu'il n'existait aucune fonction permettant de désynchroniser un objet. Cependant, cette fonction sera implémentée dans la prochaine version (v 1.5) de MiddleVR. Le support nous a donc donné accès à une version beta qui a résolu notre problème.

INTELLIGENCE ARTIFICIELLE

Nous allons maintenant passer à l'une des deux majeures parties de ce stage: l'intelligence artificielle des personnages. Comme nous l'avons mentionné dans le cahier des charges, les personnages devaient avoir l'air le plus réaliste possible dans leurs comportements. Nous les avons doté d'une sorte d'intelligence artificielle afin qu'ils puissent se déplacer et interagir avec l'environnement. Pour cela, nous avons crée le script "IA_Humanoide.cs" que nous avons placé sur chacun des personnages.

Ce script est composé de deux parties: une première partie interaction avec l'environnement, et une deuxième partie navigation dans la scène.

Afin d'interagir avec l'environnement, nous avons crée un système d'état aux personnages. C'est à dire qu'ils agiront de façon différente en fonction de l'état dans

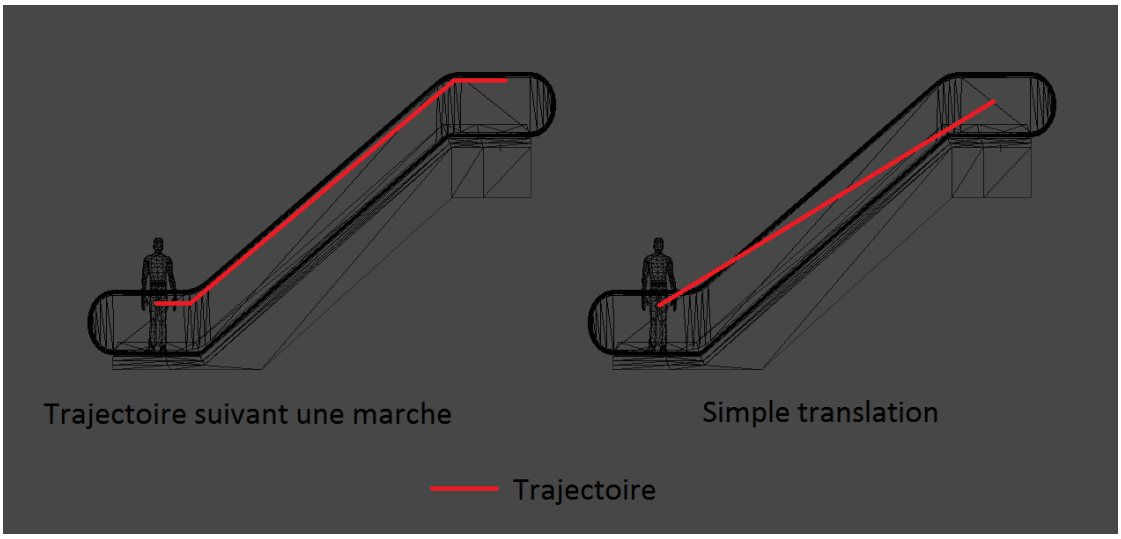
[8] Motion Capture: technique permettant d'enregistrer les positions et rotations d'objets ou de membres d'êtres vivants, pour en contrôler une contrepartie virtuelle sur ordinateur.

lequel ils se trouvent. Nous avons défini plusieurs états (5 au total) que nous allons détailler ci-dessous.

- L'état "Nothing": cet état est simplement là pour indiquer au personnage qu'il n'interagit avec aucun élément et peut donc continuer à avancer sans se préoccuper de rien. Nous nous servons tout de même de cet état pour vérifier si le personnage n'entre pas en interaction avec des éléments extérieurs comme un escalator ou encore un magasin. Si une interaction est détectée, l'état du personnage change et son comportement s'adaptera dès la prochaine image.

- L'état "Magasin": dès qu'un personnage s'approche d'un magasin, celui-ci possède une chance sur 4 de s'arrêter devant quelques secondes afin de l'observer et d'imiter des personnes en train de faire leurs courses. Nous avons matérialisé les magasins par de simple GameObjects invisibles comportant simplement une zone de collision définissant là où les personnages vont s'arrêter et contempler le magasin. Mais un problème se pose : cette chance sur 4 nécessite en théorie l'utilisation d'une fonction aléatoire. Et c'est la où nous avons dû être astucieux afin d'éviter de synchroniser de nouvelles variables, et par conséquent empêcher d'utiliser trop de ressources réseau. En effet, nous avons mis en place un système d'identifiant (ID). Chaque personnage possède donc un ID unique qui lui est attribué dès sa création. On peut considérer qu'il s'agit de leur ordre d'apparition dans la simulation : le premier personnage ayant l'ID n°1, le deuxième l'ID n°2, et ainsi de suite. Une fois cet ID attribué, pour reproduire la chance sur 4 de s'arrêter devant un magasin, nous divisons l'ID du personnage et regardons si le reste est égal à 0. Si c'est égal à 0, cela veut dire que l'ID est un multiple de 4, et donc que uniquement un personnage sur 4 va s'arrêter devant le magasin. Le seul défaut de cette méthode réside dans le fait que ce sont toujours les mêmes personnages qui vont s'arrêter devant les magasins, mais cela n'est pas très gênant.

- L'état "Escalator": si un personnage entre dans l'état escalator, celui ci s'arrête immédiatement de marcher et prend l'escalator. Afin de rendre le mouvement des personnages le plus réaliste possible, ces derniers suivent la trajectoire de la marche sous leurs pieds plutôt que de réaliser une simple translation entre le point d'entrée et de sortie de l'escalator. Ainsi nous avons placé les points d'entrées des escalators au dessus des marches. Cela nous permet de récupérer facilement à l'aide d'un Raycast la marche sous les pieds des personnages et d'appliquer les déplacements de la marche à ceux ci.

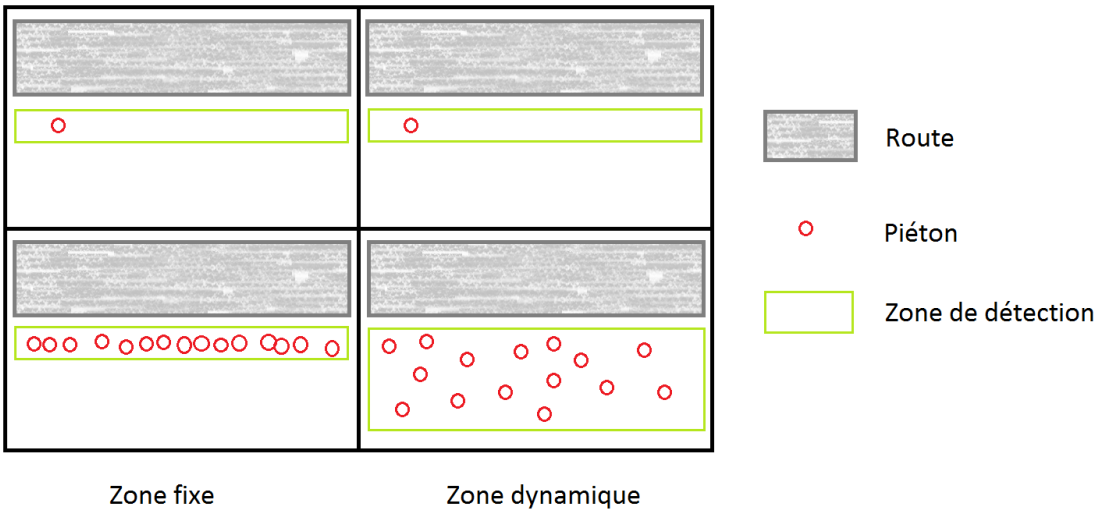


(Schema des deux choix possible de trajectoire pour monter un escalator)

- L'état "FollowPath": Permet au personnage de suivre un chemin prédéfini (chaque scène possède son propre chemin prédéfini). Cet état nous permettra par la suite de simuler une foule suivant ce chemin.

- L'état "WaitForCross": Il s'agit du dernier état que les personnages peuvent avoir. Il leur permet d'attendre avant de traverser une rue pendant que les voitures passent. Cet état sauvegarde l'état précédent afin que le personnage ne change pas d'état en attendant de traverser. Ce dernier état est déclenché de la même manière que pour les magasins : un GameObject vide avec une zone de collision à l'emplacement du passage piéton. Les seules différences avec les magasins sont :

- que le moment où les personnages sont "relachés" ne dépend plus d'un temps aléatoire mais du passage au vert des feux de circulation.
- que la zone s'agrandie au fur et à mesure que les piétons arrivent pour traverser. De cette façon nous évitons une agglutination de personnes et cela augmente un peu plus le réalisme.



(Schéma de répartition des piétons, comparaison entre une zone fixe et une zone dynamique)

Toutes ces interactions servent à augmenter le réalisme de la scène, et de faire en sorte que les personnages soient "conscients" des éléments qui les entourent. Mais avant d'interagir avec l'environnement, les personnages doivent tout d'abord se déplacer dans celui-ci. Nous allons donc aborder la question des déplacements.

Pour que les personnages se déplacent dans la scène, nous avons placé des points de passage. De cette façon, les personnages se déplacent de point en point de façon aléatoire afin qu'ils aient l'air de savoir où ils vont. Pour renforcer cette idée, et éviter qu'ils ne fassent que des aller-retour, nous avons décidé de leur ajouter un champ de vision. Ce champ de vision va leur permettre de choisir aléatoirement leur prochain point de passage parmi ceux se trouvant à l'intérieur de ce champ. Bien sûr, si aucun point de passage ne se trouve dans le champ de vision (ce qui arrive dans les coins de la scène), nous leur attribuons dans ce cas un point de passage aléatoire. Ici, le fait que le choix des points de passage soit aléatoire ne nous pose aucun souci. En effet, la position et la rotation des personnages étant déjà synchronisées, peut importe le choix des points de passage des autres clusters, se sera toujours le choix du cluster maitre (l'écran de devant) qui sera choisi.

Une fois le point de passage choisi par le personnage, il faut le faire naviguer jusqu'à ce dernier. Pour cela, nous allons utiliser ce que l'on appelle le "PathFinding". Il s'agit de trouver comment se déplacer dans un environnement entre un point de départ et un point d'arrivée en prenant en compte différentes contraintes.

PATHFINDING

Nous arrivons maintenant sur la deuxième plus grosse partie du stage, mais également la plus intéressante pour moi puisqu'il s'agissait de quelque chose de tout nouveau. Cette partie représente également un challenge de taille permettant à la simulation d'être véritablement la plus réaliste possible. Nous allons donc découper cette partie en plusieurs sous-parties afin d'expliquer les solutions envisagées, mais aussi les solutions testées.

RECHERCHE DE SOLUTION

Nous avons dans un premier temps utilisé la solution de base proposée par Unity. Cette solution repose sur l'utilisation d'un NavMesh[9]. Une fois le NavMesh généré dans la scène, les GameObjects possédant le script "NavMeshAgent" sont capable de se déplacer en empruntant le plus court chemin d'un point A à un point B sur le NavMesh en question. L'avantage de cette solution est qu'elle est rapide à

[9] NavMesh: structure de donnée utilisée en intelligence artificielle permettant de représenter les zones d'un environnement 3D.

mettre en place, simple et efficace. Mais le gros problème est qu'elle ne permet pas aux personnages de s'éviter entre eux. Or il s'agit d'un problème extrêmement gênant lorsque deux groupes de personnes doivent se croiser dans un petit espace. Prenons un exemple: lorsque deux personnages veulent se croiser entre un mur et un pilier, aucun des deux n'aura "l'intelligence" de passer de l'autre coté du pilier et par conséquent ils se bloqueront mutuellement. Nous avons donc d'abord cherché à détecter si un autre personnage était en face de celui qui avance, et à le faire se décaler si le cas se présente. Il en a résulté un déplacement pas du tout réaliste, et cette solution ne marchait pas lorsque deux groupes de personnes se rencontraient (certains étaient entourés de personnes et se mettaient à tourner en rond).

Nous avons alors cherché du coté des solutions déjà implémentées (sous forme de Plug-in pour Unity) et en avons étudié quelques unes. Tout d'abord nous avons commencé avec "Critter AI", projet développé par Google servant de base pour comprendre comment fonctionne le PathFinding. Nous n'avons pas retenu cette solution car elle ne nous permettait pas grand chose de plus que le PathFinding de base d'Unity.

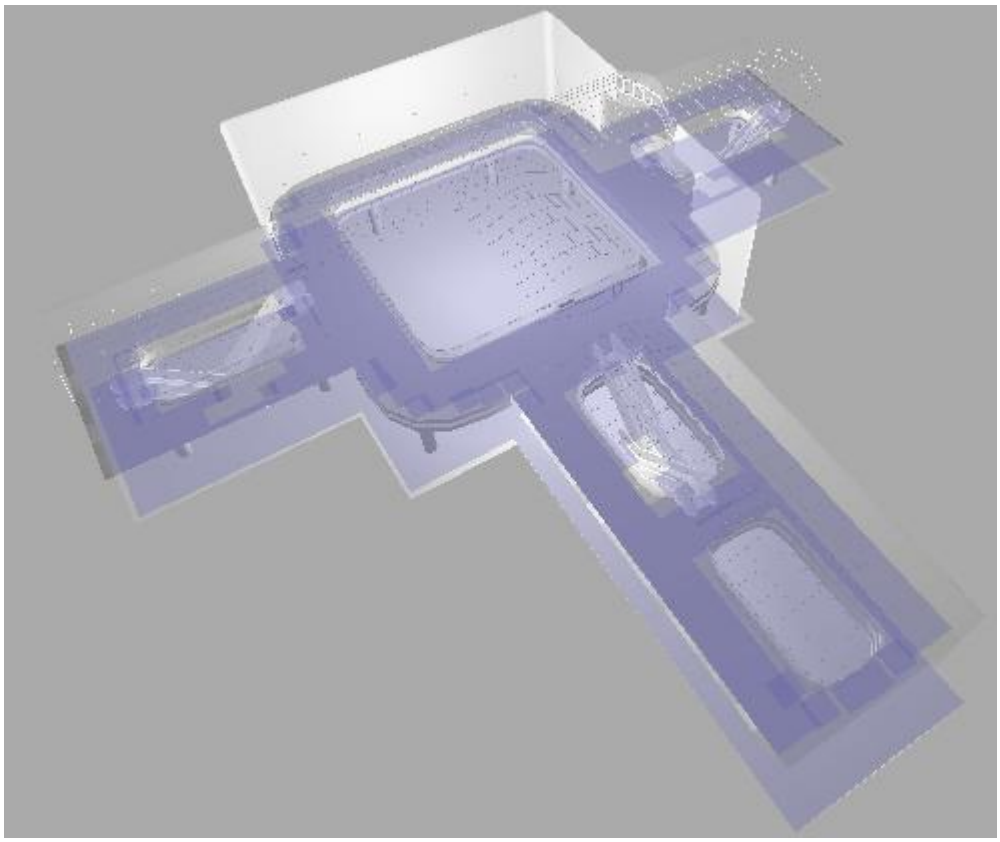
En continuant à chercher, nous avons trouvé le plug-in "RAIN", développé par la compagnie Rival Theory. Cette solution était intéressante puisqu'en feuilletant les tutoriaux, il devait nous être possible de faire s'esquiver les personnages entre eux tout en choisissant un chemin court. De plus, une version gratuite était disponible et accessible à tous. Malheureusement, ce système de PathFinding faisait complètement chuté le nombre d'images par seconde à cause du nombre de chemins à calculer en temps réel (un chemin par personnage, soit cent chemins à calculer en temps réel).

Pour finir, nous avons également étudier le "A* PathFinding Project" développé par Aron GRANBERG avec l'aide de sa communauté. Ce projet repose sur l'utilisation de l'algorithme A*. Il s'agit d'un algorithme de recherche de chemins dans un graphe entre un nœud initial et un nœud final. Malheureusement, ce projet n'était pas gratuit, nous ne l'avons donc pas choisi, mais nous nous en sommes fortement inspiré afin de réaliser notre propre solution de PathFinding.

ETUDE D'UN PREMIER ALGORITHME DE PATHFINDING

Après avoir rechercher des solutions existantes sans succès, nous avons décidé de réaliser notre propre solution de PathFinding. Le principal avantage de ce choix est que le résultat devrait répondre parfaitement à notre problématique, au prix d'un important temps de développement.

Nous avons donc conçu un premier algorithme de PathFinding. Pour ce faire, nous avons recrée un NavMesh à partir de carrés 2D mis bout à bout quelques centimètres sous le sol.



(Notre NavMesh dans centre commercial apparaissant en bleu)

Une fois le NavMesh créé, nous avons mis un script sur chaque carré afin qu'ils détectent au début de la simulation les carrés alentours afin d'établir un graphe. Nous avons également doté chaque parcelle (carré) d'un poids. En effet, le poids de base d'une case est de 1, et ce poids sera augmenté de 4 pour chaque personnage dessus. Cela va nous servir dans l'algorithme afin de détecter les personnages sur le chemin calculé. Une fois cette étape réalisée, nous allons nous occuper de l'algorithme qui va nous permettre de trouver un plus court chemin entre deux points tout en essayant d'éviter les autres personnages.

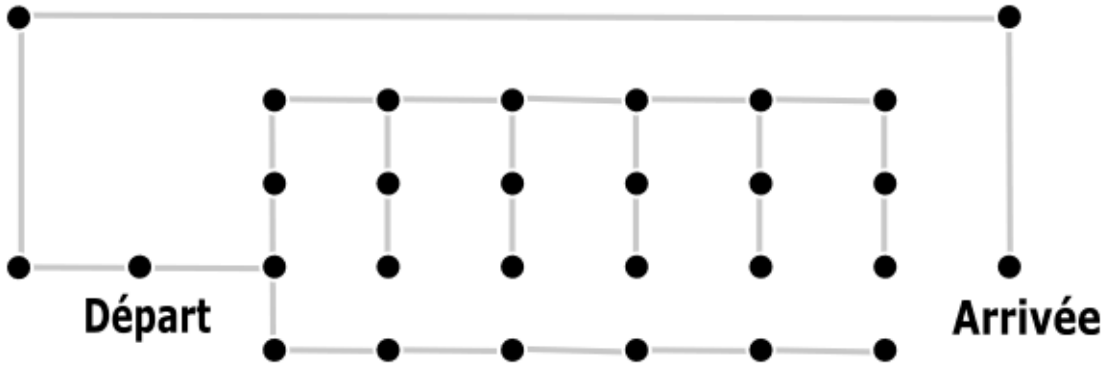
Pour cela, nous nous sommes inspiré de l'algorithme de Dijkstra. Le but principal de cet algorithme est de rechercher LE chemin le plus court. Or nous voulions un chemin court tout en esquivant au maximum les personnages, ce qui veut dire qu'il ne s'agira pas forcement du chemin le plus court. Nous l'avons donc modifié pour répondre à ce problème de la manière suivante.

L'algorithme réalise une recherche en utilisant la pondération des cases comme une distance. Ainsi pour atteindre une case où il n'y a personne, le coût de l'opération sera de 1, alors que si une personne est sur la case, le coût de l'opération sera de 5 (1+4 en raison de la personne présente sur la case). Chaque case est enregistrée dans un tableau en fonction de son coût, et l'algorithme traitera les cases en commençant par celles où le coût est le plus faible. De cette façon, le chemin trouvé sera court, mais esquivera également les personnages sur le chemin si le coût de l'esquive est inférieur à celui du "passage en force".

Cette première version de PathFinding est néanmoins plutôt imparfaite. Elle ne permet pas totalement un calcul d'une centaine de chemin en temps réel (à cause des 100 personnes), et prend énormément de place en mémoire. Nous avons donc décidé de refaire un algorithme permettant de palier à ces problèmes.

ETUDE D'UN DEUXIEME ALGORITHME DE PATHFINDING

La deuxième version de notre PathFinding, qui est notre version finale, repose quand à elle sur l'algorithme A*. Il s'agit d'une évolution de l'algorithme de Dijkstra. En effet, cet algorithme prend en compte des informations supplémentaires telles que la direction dans laquelle il faut chercher. Ces informations permettent d'orienter la recherche afin de trouver le plus rapidement possible un chemin court sans pour autant trouver le chemin optimal. Cet algorithme n'est cependant pas à utiliser dans toutes les circonstances. En effet, si le chemin le plus court nécessite de partir dans le sens opposé du point d'arrivée, cela représente alors une perte de temps de calcul énorme. Si tel est le cas, l'algorithme de Dijkstra sera le meilleur.



(Exemple de cas ou l'algorithme A* n'est pas intéressant / source: Wikipedia)

Heureusement, dans notre cas, le chemin le plus court est quasiment tout le temps dans la direction du point d'arrivée.

Nous avons donc retranscrit l'algorithme A* dans notre programme en le modifiant légèrement. L'algorithme se déroule de la façon suivante : il commence d'abord par regarder les cases adjacentes à la case de départ, puis les rangent dans un tableau trié en fonction d'un quotient égal à la distance entre la case d'arrivée et la case regardée auquel on rajoute la pondération de cette case. Une fois toutes les cases adjacentes rangées dans le tableau, on supprime la case en cours du tableau et on prend la case arrivant en tête de celui-ci, c'est à dire la case la plus proche théoriquement du point d'arrivée. Le fait d'ajouter la pondération des cases à leur distance du point d'arrivée permet de les rendre moins intéressantes pour l'algorithme: il pensera que ces cases sont plus éloignées que ce qu'elles ne le sont réellement.

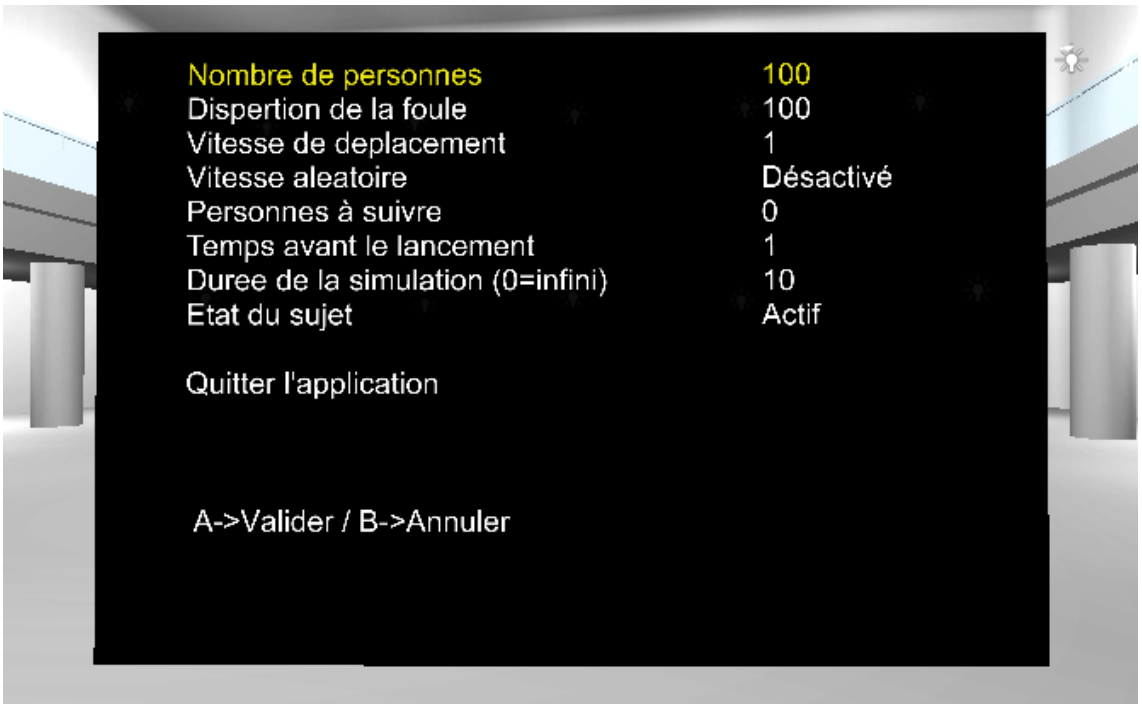
Résultat de ce deuxième algorithme : nous sommes passés de plusieurs tableaux pour stocker les cases à explorer à un seul et unique tableau trié. Nous

gagnons également du temps de calcul puisque la recherche est dorénavant orientée vers le point d'arrivée, n'exécutant non plus une recherche en cercle, mais une recherche en ligne droite. Nous avons donc un algorithme capable de calculer un chemin court en évitant au maximum les personnages (prenant des chemins alternatifs s'il y a un attroupement de personnages), le tout une centaine de fois en temps réel et en ne prenant que très peu de ressources.

Le dernier petit souci à régler concernant ce système de PathFinding était celui du lissage de chemin. En raison du NavMesh composé de carrés, le chemin trouvé possède la forme d'un zigzag, allant du centre d'une case à une autre. Pour cela, nous avons décidé d'appliquer une certaine inertie des mouvements précédents : on divise par 20 la direction où le personnage veut aller, on l'ajoute à la direction en cours prise par le personnage, et on normalise le tout pour éviter que le personnage prenne trop de vitesse. Cette solution n'a absolument rien de physique et est uniquement arbitraire. Elle permet tout de même un bon résultat, évitant ainsi les changements brusques et incessants de direction des personnages.

INTERFACE

Pour finir ce projet, nous avons réalisé une interface graphique afin que l'on puisse régler les paramètres de la simulation facilement et avec la manette. Nous n'avons pas utilisé le GUI[10] prédéfini d'Unity puisque celui-ci ne supporte pas l'affichage 3D du CAVE. Nous avons utilisé des textes en 3D que nous avons placé devant un fond noir afin de mieux discerner les lettres par rapport au fond.



(Interface permettant de modifier les paramètres de la simulation)

[10] GUI: Interface Graphique

RESULTAT

Au terme de ce stage, nous avons, du coté de la programmation, rempli le cahier des charges en livrant une application fonctionnelle. Du coté de la modélisation, le travail est toujours en cours afin de texturer et meubler les environnements.

Nous allons maintenant nous tourner vers l'utilisation de l'application, et voir le cheminement d'un test.

DEROULEMENT D'UN TEST

Premièrement, nous avons une phase de réglage. Nous choisissons tout d'abord une scène afin de charger l'environnement dans lequel notre foule va prendre vie. Une fois le choix effectué, la scène se charge et le menu des paramètres s'affiche automatiquement.

Après avoir choisi nos paramètres, la foule se crée mais ne s'anime pas. Les personnages à suivre sont affichés en surbrillance avec le message "ICI" au dessus de leur tête afin de mieux les repérer.

(Exemple d'affichage des personnages cibles)

Après la fin du temps d'initialisation (temps pendant lequel le sujet doit repérer les personnages), la foule s'anime et le sujet suit les cibles des yeux pendant un certain temps défini dans les paramètres.

Une fois le temps écoulé, la foule s'arrête et l'identifiant de chaque personnage s'affiche au dessus de sa tête, afin que le sujet puisse donner une réponse facilement à l'examinateur.

Puis pour finir, les résultats s'affichent en remettant en surbrillance les personnages cibles afin de vérifier si les réponses données par le sujet sont correctes.

Une fois le test fini, l'utilisateur peut relancer un autre en changeant les paramètres s'il le souhaite, ou arrêter la simulation.

SUGGESTION / AMELIORATION

Ce projet est néanmoins très imparfait puisqu'il reste quelques bugs à éliminer. Mais il serait aussi intéressant d'ajouter de nouvelles fonctionnalités. Par exemple, nous pourrions ajouter assez facilement un cinquième écran afin d'afficher les résultats, contrôler la simulation en temps réel ou encore la paramétrer.

L'optimisation serait aussi un vrai plus pour la simulation, surtout au niveau des modèles 3D (possédant trop de points) et du PathFinding (étant encore trop lent à notre goût).

L'ajout d'interaction entre les personnages serait également un vrai plus en terme de réalisme. La réalisation de groupe de personnes pourrait également augmenter la crédibilité de la foule.

Mais la fonctionnalité que nous trouvons la plus importante à ajouter est une fonction de calcul des statistiques, enregistrant les réponses du sujet sur un fichier que les chercheurs pourraient ensuite directement utiliser avec leurs logiciels (MathLab par exemple) pour les étudier et les interpréter.

CONCLUSION

Pour conclure, ce stage était pour moi un véritable challenge pour plusieurs raisons. C'était l'occasion de travailler sur une nouvelle technologie incroyablement fascinante: le CAVE, de comprendre son fonctionnement et apprendre à l'utiliser convenablement afin d'arriver à rendre un projet fonctionnel. Ce stage m'a également permis d'aborder des sujets vus en cours mais que je n'avais encore pas eu l'occasion de mettre en pratique, comme le PathFinding reprenant les cours de Graphes.

J'ai également pu prendre part à la gestion d'une équipe, et à l'organisation d'un projet à moyen terme durant ces trois mois de stage. J'ai également pu avoir un premier contact avec le monde du travail en réalisant ce projet dans un temps donné, mais aussi grâce à la collaboration avec une entreprise (MiddleVR).

Enfin, j'ai pu apprécier le cadre de vie agréable et cosmopolite de la ville de Montréal. Le mode de vie des Québécois repose sur la confiance envers les autres et dans le monde du travail, je pense que cela se traduit par une meilleure productivité.

Ce stage m'aura donc permis de m'épanouir pleinement au sein d'une entreprise et de découvrir un autre pays et ses coutumes.