Université de Montréal

Une approche multi-agents pour le développement d'un jeu vidéo

par
Guillaume Asselin

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Juin, 2013

RÉSUMÉ

Un système multi-agents est composé de plusieurs agents autonomes qui interagissent entre eux dans un environnement commun. Ce mémoire vise à démontrer l'utilisation d'un système multi-agents pour le développement d'un jeu vidéo. Tout d'abord, une justification du choix des concepts d'intelligence artificielle choisie est exposée. Par la suite, une approche pratique est utilisée en effectuant le développement d'un jeu vidéo. Pour ce faire, le jeu fut développé à partir d'un jeu vidéo mono-agent existant et modifié en système multi-agents afin de bien mettre en valeur les avantages d'un système multi-agents dans un jeu vidéo. Le développement de ce jeu a aussi démontré l'application d'autres concepts en intelligence artificielle comme la recherche de chemins et les arbres de décisions. Le jeu développé pour ce mémoire viens appuyer les conclusions des différentes recherches démontrant que l'utilisation d'un système multi-agents permet de réaliser un comportement plus réaliste pour les joueurs non humains et bien plus compétitifs pour le joueur humain.

Mots clés: Intelligence artificielle, Jeux vidéo, Système multi-agents, Recherche de chemins, Arbre de décisions.

ABSTRACT

A multi-agent system is composed of several autonomous agents that interact with each other in a common environment. This thesis aims to demonstrate the use of a multi-agent system for the development of a video game. First, a justification of the artificial intelligence's concepts used in this master's thesis is exposed. Subsequently, a practical approach is used in developping a video game. To do this, the game was developed from an existing single-agent video game and modified into a multi-agent system in order to properly highlight the benefits of a multi-agent system in a video game. The development of this game also demonstrate the application of other concepts in artificial intelligence such as pathfindinig and behaviour trees. In summary, the use of a multi-agent system has achieved a more realistic behavior for the non-human players and a more competitive gameplay for the human player.

Keywords: Artificial intelligence, Video games, Multi-agents system, Pathfinding, Behaviour trees.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES

BDI	Belief, Desire, Intentions
BT	Behaviour tree
FPS	First-person Shooter
IA	Intelligence Artificielle
NES	Nintendo Entertainment System
PC	Personal Computer
SMA	Système Multi-Agents

NOTATION

REMERCIEMENTS

Je tiens à remercier Monsieur Roger Nkambou, professeur à l'UQAM (Université du Québec à Montréal), ainsi que Monsieur Claude Frasson, professeur à l'Université de Montréal, pour leur cosupervision respective lors de ma maitrise en informatique.

Je voudrais aussi remercier le programme SEPICS (Student Exchange Program in Intelligents Computers Systems) et Monsieur Mike Eboueya de l'Université de La Rochelle pour m'avoir permis de réaliser la première version de ce jeu ainsi que pour m'avoir donné une formation de base en programmation de jeux vidéo.

Je tiens à remercier ma copine Charlotte ainsi que mes parents pour le soutien précieux qu'ils m'ont apporté tout au long de cette maîtrise. Je n'aurais jamais pu y arriver sans leur aide.

Finalement, je voudrais remercier le studio de jeux vidéo Budge Studios pour lequel je travaille qui m'a permis de consacrer beaucoup de temps à la rédaction de ce mémoire.

CHAPITRE 1

INTRODUCTION

L'industrie du jeu vidéo est un domaine de l'informatique en pleine croissance. En 2011, cette industrie a généré des revenus de plus de 25 milliards de dollars US [4]. Les plateformes de jeux se multiplient résultant en de nouvelles catégories de jeu comme les jeux mobiles.

De nos jours, les compagnies oeuvrant dans le domaine doivent sans cesse trouver de nouveaux moyens pour captiver leur auditoire. La créativité des développeurs amène une plus grande variété de jeux afin de satisfaire une clientèle qui est elle aussi de plus en plus variée. Les compagnies cherchent toujours à repousser l'état de l'art afin de dépasser les concurrents. Beaucoup d'efforts et d'argent sont investis dans la création d'un jeu vidéo afin qu'il se démarque de ses compétiteurs. Les compagnies de jeux vidéo réalisent que les enjeux reliés à la production d'un jeu ne sont pas seulement de créer de beaux graphismes ou bien d'avoir une histoire intéressante [2]. En effet, cette industrie réalise qu'une intelligence artificielle (IA) réaliste et semblable à l'être humain est un ingrédient très important du jeu. Nous assistons aussi à une croissance du jeu en ligne [4]. Les joueurs préfèrent rejoindre des communautés de jeu en ligne et échanger avec d'autres joueurs plutôt que de jouer seul. Les jeux d'équipe sont donc très populaires en ligne, mais aussi hors ligne lorsque le joueur joue contre plusieurs adversaires contrôlés par sa console. C'est donc la console qui s'occupe de diriger les adversaires du joueur.

Pour donner à l'utilisateur l'impression qu'il joue contre de véritables personnes, l'IA du jeu doit être la plus fidèle à celle d'un être humain possible. Si l'IA est trop parfaite, le joueur pourra être frustré et verra le jeu comme étant trop difficile, tandis que si elle est trop stupide, le joueur se lassera de toujours gagner. Une IA bien équilibrée est un aspect très important dans le développement d'un jeu vidéo moderne[14].

Jusqu'à ce jour, la grande majorité des jeux d'équipe utilisent une IA plus individualiste que collective, c'est-à-dire que les actions des joueurs sont choisies seulement en fonction des percepts et de l'environnement du personnage plutôt qu'en fonction des autres personnages de son équipe [22]. Dans les jeux d'équipe traditionnels, la communication entre les membres de l'équipe est très importante [18]. Une bonne communication permet une meilleure coordination entre les coéquipiers et donc de meilleures chances de vaincre l'adversaire.

En IA, on peut représenter une équipe par un système multi-agents (SMA). Un SMA est composé de plusieurs agents individuels, les membres de l'équipe. Il serait intéressant d'intégrer la coopération dans un SMA et d'intégrer ce système à un jeu vidéo afin d'améliorer l'IA. Cette amélioration pourrait bénéficier les productions de jeux vidéo et ainsi mieux répondre aux attentes des joueurs.

Ce mémoire démontre l'utilisation pratique de concepts d'IA de haut niveaux qui sont plutôt abstraits, comme par exemple la recherche de chemins et les système multi-agents. Les concepts d'IA vus dans ce mémoire sont déjà utilisés dans les productions de jeux vidéo. La contribution scientifique de ce mémoire n'est donc pas de démontrer

si un jeu vidéo peut être développé à partir des concepts d'IA vus dans ce mémoire, mais plutôt comment le développer de façon à ce que le joueur humain est la meilleure expérience de jeu. La prémisse de base étant que l'IA dans les jeux apporte un plus grand réalisme au niveau des actions entreprises par les personnages du jeu ainsi qu'une meilleure expérience ressenti par le joueur humain. Une autre contribution majeure de ce mémoire est de réussir à modéliser un système multi-agents capable de rivaliser avec le joueur humain et que ce dernier croit qu'il joue contre de vrais adversaires humains.

1.1 Problématique

Ce mémoire de maitrise vise à répondre à la problématique suivante :

Comment intégrer un système multi-agents dans le développement de jeux vidéo ?

1.2 Méthodologie

Afin de répondre à la problématique posée, nous avons étudié les modèles acutels de système multi-agents ainsi que des jeux vidéo existants pouvant être adaptés en SMA. Par la suite, un jeu vidéo existant a été modifié afin d'utiliser un SMA.

Plusieurs itérations de ce jeu ont été développées afin de raffiner le modèle. La version finale comprend une architecture multi-agents complète représentant l'équipe s'opposant au joueur.

1.3 Aperçu du mémoire

Ce mémoire est composé de quatre parties. La première partie donne un aperçu général des jeux vidéos avec un historique de ceux-ci et une présentation de l'industrie. Ensuite, les différentes techniques d'IA utilisées dans le développement de jeux vidéo sont décrites. Ce chapitre donne un aperçu général de ces techniques et permet de comprendre les concepts fondamentaux de l'IA appliquée aux jeux vidéo.

La deuxième partie du mémoire porte sur le jeu qui a été développé durant cette maitrise. On y présente la version initiale du jeu sans système multi-agents puis les modifications qui ont été apportées afin de le rendre multi-agents.

La troisième partie porte sur l'expérience acquise par l'auteur en effectuant le développement d'un jeu vidéo simple, mais possédant suffisamment de complexité pour exposer les problèmes associés au développement de jeux vidéo en général. Une présentation des résultats reliés aux travaux présentés suit ce dernier chapitre.

Finalement, une conclusion répond à la problématique exposée dans ce mémoire. De plus, les avantages et inconvénients ou problèmes rencontrés sont exposés.

CHAPITRE 2

JEUX VIDÉO ET INTELLIGENCE ARTIFICIELLE

L'industrie des jeux vidéo est un domaine de l'informatique en pleine effervescence. La place qu'occupe cette industrie dans la ville de Montréal le démontre bien. En effet, de nombreuses compagnies s'y sont installées dont *Ubisoft, Electronic Arts, Eidos, Activision, Warner Bros et THQ*. La variété de styles de jeux ainsi que le développement d'appareils mobiles ont grandement contribué à la croissance accélérée de l'industrie.

Un bref historique des jeux vidéo est présenté dans ce chapitre afin d'illustrer leur évolution et de situer dans le temps le jeu développé dans ce mémoire [6].

2.1 Historique

2.1.1 Le début

Le premier jeu vidéo remonterait à l'an 1958 où le physicien William Higinbotham du laboratoire de BrookeHaven à New York inventa le jeu *Tennis for Two*, un jeu ressemblant au tennis et se jouant sur un oscilloscope. Le jeu fut créé simplement dans le but de divertir les visiteurs du laboratoire. Le joueur disposait d'un bouton pour frapper la balle et d'un variateur pour changer l'angle de projection. L'écran de l'oscilloscope affichait la trajectoire de la balle vue de côté. La trajectoire de la balle était inversée lorsque cette dernière frappait le sol. Pensant n'avoir rien inventé, Higinbotham ne fit pas breveter le

jeu.

Les premiers jeux vidéo, inventés à la fin des années 50 et dans les années 60 sont caractérisés par le fait qu'ils ont été développés dans des laboratoires ou des universités, notamment au MIT (Massachusetts Institute of Technology) et à l'Université de Cambridge par des scientifiques et qu'ils n'ont jamais été commercialisés.

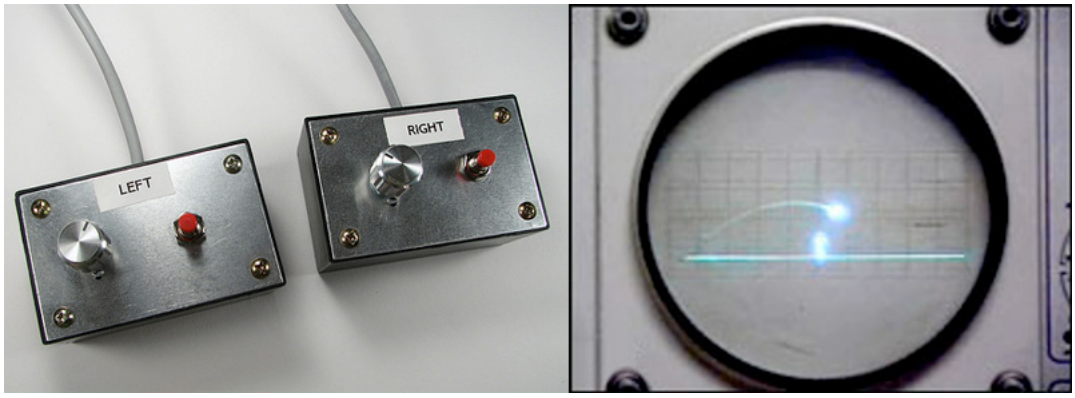

Figure 2.1 – *Tennis for two*, premier jeu électrique avec affichage

2.1.2 L'ère arcade

La période arcade a débuté dans les années 70 et se caractérise par une première tentative de commercialisation des jeux vidéo et par un premier contact aux jeux vidéo par le grand public.

En 1971, Nolan Bushnell développe le premier jeu arcade nommé *Spacewar* qui sera distribué par un fabricant de machines à sous. Le jeu n'obtiendra pas le succès escompté parce que le public le trouve trop difficile. Bushnell n'abandonnera pas et fondera *Atari* en 1972. Leur premier jeu, nommé *Pong*, obtiendra un succès monstre.

Au même moment, la compagnie *Magnavox* met sur le marché la première console de l'histoire, le *Magnavox Odissey*.

Figure 2.2 – *Magnavox Odissey*, la première console de jeux vidéo

2.1.3 L'age d'or

En 1978, la compagnie *Midway* sort le jeu *Space Invaders*, il s'agit du premier jeu gardant en mémoire le score des joueurs dans le but d'afficher les scores à battre.

La compagnie *Namco* met sur le marché *Pac-Man* en 1980, le jeu arcade le plus populaire de tous les temps.

En 1981, *J. K. Greye Software* publie *3D Monster Maze*, le premier jeu 3D. Il s'agit d'un jeu à la première personne où le joueur doit sortir d'un labyrinthe en évitant de se faire manger par un monstre.

Cette époque fait référence à l'ère des machines arcades où l'on a crée beaucoup de jeux de tous les styles et pour tous les genres.

2.1.4 Les premières consoles

Plusieurs compagnies se lancent en affaire dans le marché des consoles au début des années 80 suite aux succès du *Magnavox Odissey*. Les gens délaisseront peu à peu les systèmes arcades pour les consoles de salon puisque celles-ci permettent aux joueurs de jouer à leurs jeux d'arcades préférés dans le confort de leur maison.

Durant l'année 1985, *Nintendo* teste sa console le *Nintendo Entertainment System* (NES) à New York. Les marchands sont plutôt retissant face à la console, mais avec beaucoup de titres développés par Nintendo eux-mêmes, la console fait fureur.

La même année, le programmeur russe Alex Pajitnov invente *Tetris*, un jeu simple, mais très addictif où des pièces de différentes formes tombent du haut vers le bas. Le joueur doit emboîter ces pièces afin de créer des lignes horizontales pleines. Lorsqu'une ligne pleine est créée, elle disparaît. Si le joueur ne parvient pas à faire disparaître les lignes assez vite et que l'écran se remplit jusqu'en haut, la partie se termine.

En 1989, la console portable *Game Boy* de *Nintendo* sort sur le marché. Le jeu *Tetris* est inclus avec la console et *Nintendo* brise les records de vente d'unité.

Face au succès de *Nintendo*, *Sega* sort la *Game Gear* avec son titre vedette : *Sonic the Hedgehog* dans le but de conquérir les fans de *Mario* sur *Nintendo*.

2.1.5 Consoles modernes

Depuis le NES, les consoles n'ont pas arrêté de s'améliorer grâce à de nouvelles composantes à la fine pointe de la technologie. Étant donné que l'architecture d'une console

moderne ressemble à celle d'un ordinateur (lecteur optique, disque dur, carte graphique, etc.), les compagnies de jeux vidéo sortent généralement leurs titres sur consoles et sur ordinateur en même temps. Par exemple, le *Playstation 3* de *Sony* comprend un processeur de 3,2 GHz et une carte vidéo de 256 Mo.

Au moment d'écrire ce mémoire, les trois consoles les plus populaires sont la *Wii* de *Nintendo*, axée sur la famille avec ses contrôleurs innovateurs et ses titres moins violents, le *Xbox 360* de *Microsoft* et le *Playstation 3* de *Sony* sont tous deux très semblables au point de vue du public cible, des services et des titres. Ces deux dernières consoles s'adressent à un public plus mature et la connectivité à internet permet d'installer des applications sur sa console un peu à la façon des téléphones intelligents. Les trois consoles offrent des performances et des prix similaires.

2.2 Industrie du jeu vidéo

Tel que mentionné dans le chapitre 1, l'industrie du jeu vidéo génère plusieurs milliards de dollars par année. Les plateformes sont de plus en plus accessibles, presque chaque famille a accès à une plateforme de jeux que ce soit un PC (personal computer), une console branchée à la télévision, une console portable ou un téléphone intelligent.

Les jeux vidéo rejoignent autant les femmes que les hommes [4] et ils rejoignent aussi toutes les tranches d'âge. Il se jouent aussi de plus en plus en famille, cela permet aux parents de socialiser avec leurs enfants et beaucoup d'entre eux considèrent que les jeux vidéo peuvent aider leurs enfants à développer leurs facultés mentales.

Une plus grande variété de joueurs amènent une plus grande variété de titres. Cependant, pour se démarquer des autres jeux, les compagnies de jeux vidéo misent sur certaines caractéristiques essentielles comme :

- Le «gameplay» qui représente le ressenti du joueur ou l'interaction entre le joueur et le jeu.
- Les rendus graphiques du jeu. Pour beaucoup de joueurs, des graphiques réalistes sont une qualité importante d'un jeu.
- L'audio du jeu, comme la narration, la musique ambiante et les effets spéciaux.

On cherche toujours un bon «gameplay», une expérience positive pour le joueur. Le «gameplay» regroupe plusieurs aspects du jeu comme la difficulté, les règles du jeu, l'interaction avec les objets dans le jeu et bien sûr l'IA des ennemis. En général, c'est le «game designer» qui s'occupe du «gameplay». Il rédige un document expliquant chaque scène du jeu en détails puis c'est le travail des programmeurs d'implémenter ces comportements dans le jeu. Cela implique aussi d'évaluer plusieurs possibilités en testant sur des prototypes et en effectuant de nombreux cycles de développement du jeu.

Pour le rendu graphique du jeu, les «game designers» doivent travailler avec les artistes afin de déterminer avec eux ce qui est possible de faire avec le moteur graphique du jeu. Le moteur 3D est la composante logicielle qui s'occupe de convertir les images et objets 3D en images matricielles. La technique la plus utilisée en jeu vidéo pour faire cette transformation se nomme la «rastérisation» [3]. La rastérisation est le procédé où

les objets sont représentés sous forme de triangles, auxquels la carte graphique fait subir différentes transformations géométriques afin de les projeter sur l'écran. La carte graphique de la console s'occupe ensuite de transformer ces triangles en ensemble de pixels, puis calcule la couleur de chaque pixel. Pour déterminer la position à l'écran de chaque pixel, le pipeline est souvent le suivant :

1. On positionne chaque point de l'objet selon un point d'origine dans le modèle 3D lui-même.

2. On repositionne les points dans le monde 3D donc selon les autres modèles.

3. On repositionne les points par rapport à la position de la caméra.

Toutes ces transformations sont calculées avec le calcul matriciel.

L'audio est une composante du développement de jeu vidéo qui s'est développé rapidement, mais qui ne change pas beaucoup de nos jours. Au début des premières consoles, la musique des jeux vidéos était de simples notes. Maintenant, de la même manière que les films, les consoles supportent le *Dolby Digital* avec plusieurs canaux.

2.2.1 Contraintes de l'industrie

Le développement rapide de l'industrie amène aussi certaines contraintes. Pour rester compétitives et pour répondre à la demande de plus en plus grande et de plus en plus variée, les compagnies de jeux vidéo doivent mettre sur le marché plusieurs jeux dans la même année. Le coût d'un projet de jeu vidéo étant de plusieurs millions de dollars,

la pression est grande car on doit vendre au moins un million de copies du jeu afin que le projet soit rentable. Ce risque donne lieu à un processus de création et d'innovation limité. On doit essayer d'innover ce qui existe déjà sur le marché sans prolonger le cycle de production. Généralement, les gros joueurs de l'industrie achètent les droits liés à des titres de plus petits studios de conception sous forme de licence d'utilisation commerciale dont le paiement, sous forme de redevances, est lié au succès obtenu par le jeu sur le marché [12].

2.2.2 Mode de travail par projets

Bien souvent, un projet de développement de jeu vidéo lie une équipe de conception à un éditeur de jeu. Parfois, le studio remplit ces deux fonctions. L'équipe de conception est dirigée par un producteur et à la fin de chaque projet, l'équipe est dissoute et affectée à un autre projet.

Les 3 paramètres les plus importants dans la gestion d'un projet de jeu vidéo sont :

- L'argent.
- Le temps.
- Les caractéristiques attendues du produit (l'envergure du projet).

On remarque souvent dans les projets qui n'ont pas eu le succès escompté que la clé de l'échec est causé par une combinaison de ces facteurs[12].

2.3 Intelligence artificielle

Lorsque les recherches en intelligence artificielle ont commencé, les jeux vidéo n'existaient pas encore [10]. Cependant, l'intérêt d'appliquer les concepts d'IA aux jeux était présent. On testait les concepts d'IA sur des jeux classiques comme les échecs, les dames, le tic-tac-toe, etc. Les chercheurs commencèrent donc à programmer des ordinateurs capables de jouer à ces jeux.

En 1950, Claude Shannon inventa le premier programme pouvant jouer aux échecs. Cinquante ans plus tard, soit en 1997, le champion d'échec Gary Kasparov perdait un match d'échecs contre *DEEP BLUE*. À l'époque *DEEP BLUE* pouvait évaluer 200 millions de positions à la seconde, alors qu'un joueur humain peut seulement analyser 2 positions à la seconde [20]. Cette victoire de *DEEP BLUE* sur Kasparov démontra la puissance du cerveau humain et aussi la possibilité de créer des machines qui peuvent rivaliser avec l'être humain.

L'intelligence artificielle dans les jeux vidéo est utilisée de plusieurs façons. Une première application de celle-ci est la recherche de chemins. Ce concept permet à un ou plusieurs personnages de se déplacer d'un point A à un point B en tenant compte du terrain qui peut parfois être en deux dimensions, des obstacles dynamiques et du champ de vision du personnage.

L'intelligence artificielle est aussi utilisée afin de balancer le niveau de difficulté du joueur humain. Généralement, avec de la pratique, un jouer humain devrait s'améliorer à un jeu avec le temps. En augmentant la difficulté au fur et à mesure que le joueur

s'améliore, le joueur humain ne se lassera pas et risque de jouer plus longtemps. Au contraire, un joueur humain qui vient de commencer à jouer ne sera peut-être pas très bon. Le jeu devra alors diminuer son niveau de difficulté afin de ne pas le frustrer.

Récemment, un autre concept de l'IA a fait son apparition en jeux vidéo, soit celui de l'apprentissage-machine. Dans ces jeux, les personnages peuvent s'adapter au joueur humain en observant son comportement et lui proposer des choix adaptés à son profil. Même si ces choix sont pris parmi un échantillon restreint. Ils donnent l'illusion d'intelligence pour le joueur humain. Un bon exemple de ce type d'IA est le jeu *Black and White* développé par *Lionhead studios*. Dans ce jeu, le joueur est dieu. Le but du jeu est de se faire obéir par le plus grand nombre de personnages. Pour convertir des villageois, le joueur peut soit utiliser la bonté ou la peur. Compte tenu des choix que le joueur humain fera dans le jeu (bon ou mauvais), les situations ne seront pas les mêmes et seront adaptées au style de jeu du joueur.

Finalement, l'usage le plus courant est le contrôle de personnage. Puisque beaucoup de jeux se jouent à plusieurs et que l'on est bien souvent seul devant sa console, le jeu contrôle les autres joueurs en tentant le mieux possible de recréer un comportement humain. L'usage de l'IA pour contrôler un personnage peut être plus ou moins avancé selon les besoins. Généralement c'est l'IA qui effectuera la prise de décisions. Cette dernière sera basée selon différents facteurs qui seront présentés à la section 2.3.1.

Dans les jeux à plus d'un joueur, on parlera de système multi-agents où plusieurs agents interagissent dans un environnement. Nous verrons plus en détail à la section

2.3.3 les différents types de système multi-agents.

Le concept d'agent en IA est très important pour les jeux vidéo parce qu'il englobe les autres concepts vus précédemment. Dans un jeu de stratégie en temps réel où plusieurs unités se déplacent en même temps. Un bon agent ne sera pas efficace s'il ne peut pas se déplacer. De plus, un moyen de balancer la difficulté d'un jeu pourrait être de forcer les agents jouant contre le joueur humain à prendre de mauvaises décisions. Puisqu'un système multi-agents est constitué de plusieurs agents, la qualité de ces derniers influencera la force du système. On peut donc rapatrier les différents concepts d'IA sous le concept d'agent.

2.3.1 Agents

Un agent est une entité qui perçoit et agit [23]. Tout au long de ce mémoire, la notion d'agent représentera un personnage dans le jeu contrôlé par l'ordinateur. L'intelligence artificielle dans les jeux vidéo est encore à un stade de premier niveau, mais ce domaine avance très rapidement. Actuellement, les jeux qui sortent sur le marché utilisent des techniques d'IA de plus en plus avancées qui sont élaborées lors du «game design». En quinze ans, nous sommes passés d'un agent prenant la meilleure décision en fonction des probabilités à un agent prenant une décision en fonction de son environnement et de ses émotions.

La notion d'agent est un concept central dans le développement de l'IA dans les jeux vidéo. Si un agent peut modifier son comportement en fonction de ce que le joueur fait,

le jeu devient beaucoup plus réaliste et immersif. Un style de jeu vidéo qui se prête bien aux agents intelligents est les jeux de tir à la première personne ou en anglais «first-person shooter» (FPS). Les FPS permettent de démontrer de vraies stratégies militaires qui sont utilisées par nos corps policiers [16]. Même si les ennemis agissent comme étant des militaires bien entraînés, le joueur, lui, n'en est souvent pas un et agit souvent de façon imprévisible, ce qui est bien difficile à simuler.

Il existe plusieurs manières de concevoir des agents, mais peu importe l'architecture adoptée, un agent peut toujours être vu comme une fonction liant ses perceptions à ses actions. Plus précisément, un agent perçoit l'environnement à l'aide de ses capteurs et il agit sur son environnement à l'aide de ses effecteurs. Ce qui différencie les différentes architectures d'agents, c'est la manière dont les perceptions sont liées aux actions.

Les auteurs Russel et Norvig [19] regroupent les architectures d'agents en quatre types, à savoir :

- Les agents réactifs
- Les agents conservant une trace du monde
- Les agents délibératifs
- Les agents BDI

2.3.1.1 Agents réactifs

Comme son nom l'indique, un agent réactif ne fait que réagir aux changements qui surviennent dans l'environnement. Autrement dit, un tel agent ne fait ni délibération ni planification, il se contente simplement d'acquérir des perceptions et de réagir à celles-ci en appliquant certaines règles prédéfinies. Étant donné qu'il n'y a pratiquement pas de raisonnement, ces agents peuvent agir et réagir très rapidement.

Dans certains cas, il vaut mieux réagir rapidement sans trop penser. Lorsque l'action se déroule rapidement, ce sont parfois nos réflexes qui dictent nos actions. Ceci permet de focaliser notre attention sur une autre tâche pendant que nos réflexes s'occupent de la tâche apprise. C'est aussi le cas lorsque nous effectuons des tâches routinières. Nous avons déjà évalué auparavant le pour et le contre d'une certaine action alors nous n'avons pas besoin de réévaluer cette action si les conditions sont semblables. Par exemple, on ne se demande plus s'il est pertinent d'amener son parapluie s'il annonce de la pluie.

Ce type d'agent agit uniquement en se basant sur ses perceptions courantes. Il utilise un ensemble de règles prédéfinies, du type «si condition alors action» pour choisir ses actions. Ces règles permettent d'avoir un lien direct entre les perceptions de l'agent et ses actions. Le comportement de l'agent est donc très rapide, mais peu réfléchi. À chaque fois, l'agent ne fait qu'exécuter l'action correspondant à la règle activée par ses perceptions.

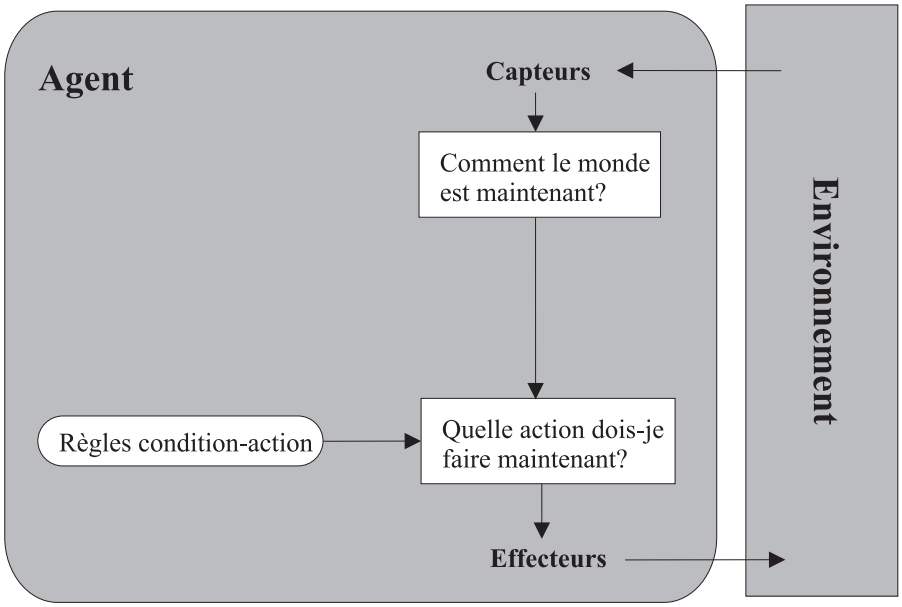

Figure 2.3 – Schéma d'un agent à réflexes simples [19].

La figure 2.3 montre l'architecture d'un agent à réflexes simples. Les rectangles représentent l'état interne de l'agent dans son processus de décision et les ovales représentent les informations qui sont utilisées dans le processus. L'agent se bâtit une représentation du monde à l'aide de ses perceptions lui venant de ses capteurs. Par la suite, il utilise ses règles pour choisir l'action qu'il doit effectuer selon ce qu'il perçoit de l'environnement.

2.3.1.2 Agents conservant une trace du monde

Un problème important des agents réactifs est qu'ils ne peuvent pas différencier un état transitoire d'un nouvel état. Par exemple, si un agent réactif est chargé de suivre un autre agent et que ce dernier se cache derrière un mur, l'agent réactif abandonnera sa

filature et retournera à son état de départ au lieu d'attendre que l'agent poursuivi sorte de sa cachette. Étant donné que l'agent n'a pas gardé en mémoire le fait que l'agent qu'il poursuivait est passé derrière le mur, il ne peut pas savoir si l'agent est derrière le mur ou qu'il n'est plus là.

Le problème que nous venons de mentionner survient parce que les capteurs de l'agent ne fournissent pas une vue complète du monde. Pour régler ce problème, l'agent doit maintenir des informations internes sur l'état du monde dans le but d'être capable de distinguer deux situations qui ont des perceptions identiques, mais qui, en réalité, sont différentes. Pour que l'agent puisse faire évoluer ses informations internes sur l'état du monde, il a besoin de deux types d'information. Tout d'abord, il doit avoir des informations sur la manière dont le monde évolue, indépendamment de l'agent. Ce dernier doit avoir ensuite des informations sur la manière dont ses propres actions affectent le monde autour de lui.

On peut voir, à la figure suivante, la structure d'un agent conservant une trace du monde. Il utilise ses informations internes (état précédent du monde, l'évolution du monde et l'impact de ses actions) pour mettre à jour ses perceptions actuelles. Par la suite, il choisit ses actions en se basant sur cette nouvelle perception du monde qui l'entoure.

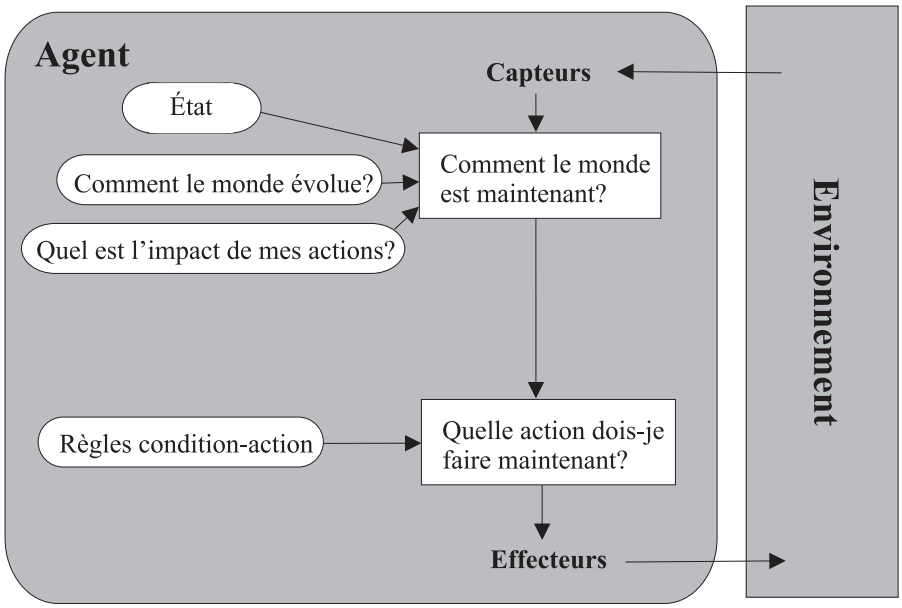

Figure 2.4 – Schéma d'un agent conservant une trace du monde [19].

2.3.1.3 Agents délibératifs

Les agents délibératifs, en plus de conserver une trace du monde, sont guidés par un but ou une fonction d'utilité afin de choisir quelle action est plus profitable d'effectuer.

Les buts représentent des situations désirables pour l'agent. L'agent peut donc combiner les informations sur ses buts avec les informations sur les résultats de ses actions pour choisir les actions qui vont lui permettre d'atteindre ses buts. Cela peut être facile lorsque le but peut être satisfait en exécutant seulement une action, mais cela peut aussi être beaucoup plus complexe si l'agent doit considérer une longue séquence d'actions avant d'atteindre son but. Dans ce dernier cas, il doit utiliser des techniques de planification pour prévoir les actions devant l'amener à son but. Bien entendu, l'agent

raisonnant sur ses buts prend, en général, beaucoup plus de temps à agir qu'un agent réactif. Il offre en revanche beaucoup plus de flexibilité. Par exemple, si nous voulions changer de destination, il faudrait changer toutes les règles de l'agent réactif, tandis que pour l'agent ayant des buts, nous ne changerions que le but. La figure suivante illustre un agent délibératif selon ses buts.

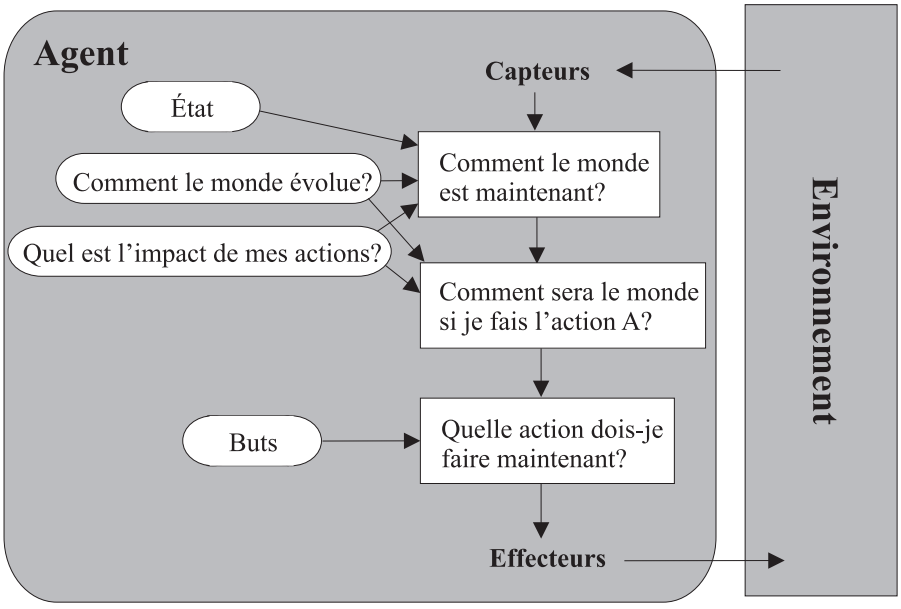

Figure 2.5 – Schéma d'un agent délibératif selon ses buts [19].

Si plusieurs chemins peuvent mener à notre but, on doit évaluer les chemins individuellement et prendre le plus prometteur selon une fonction d'utilité. Généralement, l'utilité est une fonction qui attribue une valeur numérique à chacun des chemins. Plus le chemin a une grande valeur, plus il est désirable pour l'agent. La spécification d'une fonction d'utilité permet donc à l'agent de prendre des décisions rationnelles dans deux types de situations où le raisonnement sur les buts échoue. De plus, lorsqu'il y a plu-

sieurs buts possibles, mais qu'aucun d'eux ne peut être atteint avec certitude, la fonction d'utilité permet de pondérer la chance de succès avec l'importance de chacun des buts. La figure 2.6 démontre le schéma d'un agent basé sur l'utilité.

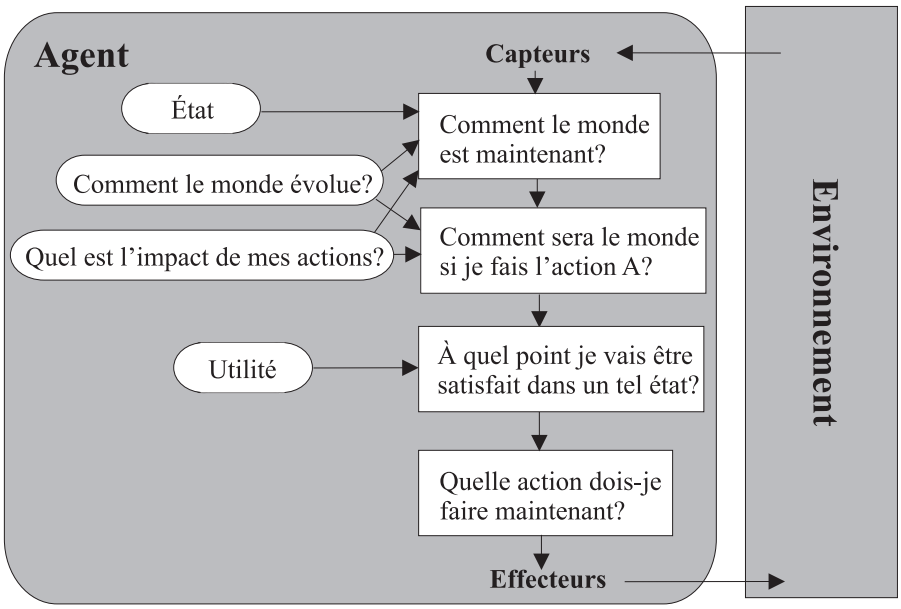

Figure 2.6 – Schéma d'un agent délibératif avec fonction d'utilité [19].

2.3.1.4 Agents BDI

BDI est un acronyme anglais pour belief, desire, intention. Les agents BDI choisissent donc leurs actions en fonction de ces trois aspects.

- Les croyances représentent les informations que l'agent possède à propos de son environnement.
- Les désirs représentent les options disponibles pour l'agent.

- Les intentions représentent le centre d'attention actuel de l'agent, c'est-à-dire les buts envers lesquels il s'est engagé et envers lesquels il a engagé des ressources.

En résumé, un agent BDI doit mettre à jour ses croyances avec les informations qui lui proviennent de son environnement, décider quelles options lui sont offertes, filtrer ces options afin de déterminer de nouvelles intentions et poser ses actions en se basant sur ses intentions.

2.3.1.5 Application aux jeux vidéo

Pour savoir quelle action un agent doit choisir étant donné l'état du jeu, dans l'industrie, on utilise généralement des machines à états finis ou des arbres de décisions [7]. Les machines à états finis ne sont pas très idéales pour exprimer l'éventail des comportements humains car elles sont trop rigides. Chaque action est gouvernée par un nombre de conditions qui doivent être satisfaites pour que l'action soit exécutée. Cette action, une fois exécutée, pourra activer d'autres conditions nécessaires à d'autres actions. Par exemple, dans un jeu de guerre, le comportement d'un agent pourrait être d'attaquer son adversaire jusqu'à ce qu'il se fasse attaquer lui-même. Il pourrait alors fuir son ennemi. La machine à états finis serait représentée par la figure suivante :

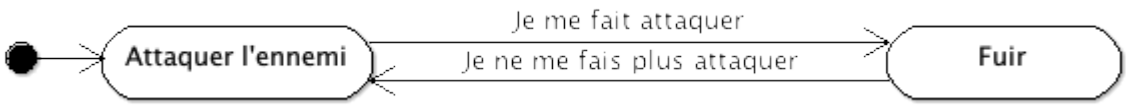

Figure 2.7 – Machine à états finis pour un agent militaire

Les machines peuvent être déterministes ou non déterministes. La plupart des jeux vidéo sur le marché utilisent des machines de type déterministe, c'est-à-dire qu'il est possible de connaitre le comportement de l'agent étant donné les conditions. On peut aussi simuler des machines non déterministes avec des nombres aléatoires pour choisir la transition d'un état à un autre. On connait tout les états, mais on ne connait pas les chemins entre les états. Le problème majeur avec les machines déterministes est qu'à force de jouer, l'utilisateur peut remarquer les conditions de transitions et ainsi prédire le comportement de l'agent ce qui rend le jeu beaucoup moins immersif. Elles sont aussi très difficiles à réutiliser ou modifier sans devoir changer plusieurs états ou plusieurs transitions. Cependant, elles sont faciles à réaliser et se prêtent bien lorsqu'il y a un petit nombre d'états à coder.

Dans l'industrie, on utilise aussi des arbres de décisions ou «behaviour trees» (BT) en anglais pour représenter des comportements humains [21]. Les arbres de décisions ont l'avantage d'être plus simple à utiliser que les machines à états finis et se prêtent plus aux applications à grande échelle [13]. Un BT est composé de noeuds et de feuilles. Les BT sont toujours parcourus en profondeur d'abord. Les noeuds déterminent quel sera le prochain enfant qui sera traversé alors que les feuilles représentent des conditions ou des actions. Il existe principalement deux sortes de noeuds :

1. Les séquenceurs vont exécuter chaque enfant de gauche à droite jusqu'à ce que ce dernier échoue. À ce moment, on sortira de cette branche et le prochain noeud exécuté sera le parent du sélecteur.

2. Les sélecteurs vont aussi exécuter chaque enfant de gauche à droite, mais à l'inverse du séquenceur, le sélecteur va exécuter un enfant différent tant que ce dernier sera un échec. Dès qu'un enfant aura réussi, on sortira de la branche pour exécuter son parent.

Comme dit précédemment, les feuilles sont principalement les conditions nécessaires à l'exécution du noeud ou la feuille suivante, ou bien l'action finale du comportement devant être déclenchée. Nous reviendrons plus en détail sur les BT à la section suivante car un BT a été utilisé pour modéliser le comportement des agents du jeu. Pour finir cette section, voici le BT de l'agent militaire.

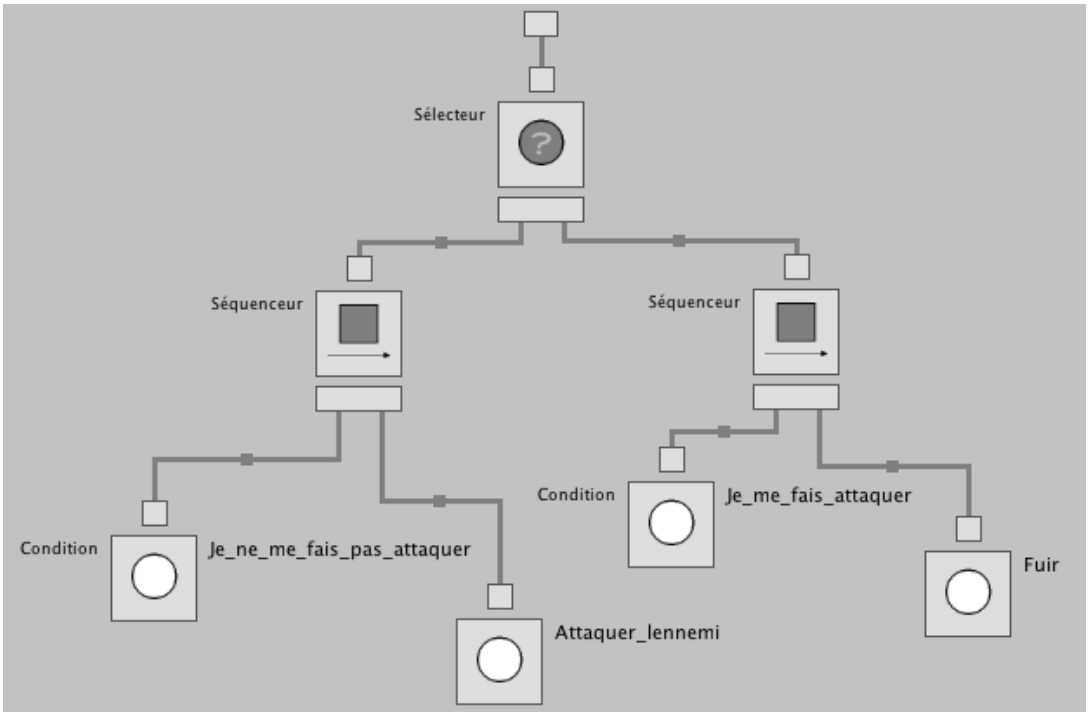

Figure 2.8 – Arbre de décision d'un agent militaire

2.3.2 Recherche de chemins

Un aspect important du développement d'un agent autonome se déplaçant dans un environnement est la recherche de chemin, ou en anglais «pathfinding». La recherche de chemins demande qu'un agent se déplace d'un point de départ à un point d'arrivée selon le plus court chemin. La recherche de chemins est très populaire dans les jeux de stratégie et dans les FPS où l'agent doit composer avec des obstacles se déplaçant dynamiquement (ennemis, coéquipiers, ressources, etc.). Il existe plusieurs stratégies de recherche pour calculer le plus court chemin. Les plus populaires sont : la recherche en profondeur d'abord, en largeur d'abord, itérative en profondeur, l'algorithme de dijkstra et l'algorithme A*. L'algorithme le plus utilisé dans l'industrie est assurément l'algorithme A* [5, 17].

L'algorithme A* est un algorithme de type «best-first», c'est-à-dire que l'algorithme choisit le chemin le plus prometteur en premier. En lui donnant un état de départ et un état d'arrivée, l'algorithme est en mesure de trouver le chemin le plus court entre ces deux états. Il utilise la fonction d'évaluation suivante pour déterminer quel sera le prochain noeud évalué [5] :

$$f(n) = g(s,n) + h(n,g) \tag{2.1}$$

Ici, *g(s,n)* est la distance entre l'état de départ s et le noeud à évaluer. *h(n,g)* est l'heuristique entre le noeud à évaluer et l'état but g. L'heuristique est une estimation du

coût minimal, il est donc très important de choisir une bonne fonction heuristique. Pour la recherche de chemins, les heuristiques généralement utilisées sont :

- La distance de Manhattan. La distance de Manhattan est facile à appliquer pour les déplacements sur une grille. La fonction d'évaluation est le coût minimum pour se déplacer à la case adjacente. En admettant que sur une grille on puisse se déplacer selon 4 directions. La fonction d'évaluation devient :

$$h(n,g) = D * (|n.x - g.x| + |n.y - g.y|) \tag{2.2}$$

- La distance diagonale. Si les déplacements en diagonale sont permis, la fonction d'évaluation est un peu différente. Par exemple, pour se déplacer de la case (0,0) à la case (2,2) on doit faire deux déplacements vers le nord et deux déplacements vers l'est selon la distance de Manhattan. Cependant, on pourrait faire seulement deux déplacements nord-est avec la distance diagonale. La fonction heuristique sera alors :

$$h(n,g) = D * \max(|n.x - g.x|, |n.y - g.y|) \tag{2.3}$$

- Finalement, si l'on peut se déplacer selon n'importe quel angle, on peut utiliser la distance euclidienne. La fonction d'évaluation devient :

$$h(n,g) = D * \sqrt{((n.x - g.x)^2 + (n.y - g.y)^2)} \tag{2.4}$$

Il existe d'autres fonctions d'évaluations, mais les précédentes sont les plus utilisées. Plus l'estimé se rapproche de la distance réelle restante à parcourir, plus vite l'algorithme A* trouvera la solution. Si la fonction d'évaluation surestime la distance réelle restante, l'algorithme n'est pas garanti de trouver le meilleur chemin. Il faut donc trouver une estimation qui est très près de la distance réelle sans la dépasser.

En pratique, pour exécuter l'algorithme A* nous avons besoin de deux listes. Une liste contenant les noeuds non visités et une autre contenant les noeuds visités. L'algorithme choisit le noeud de la liste ouverte avec le plus faible coût calculé par la fonction $f(n)$ et ajoute les noeuds voisins à la liste ouverte. Pour déterminer si un noeud voisin ira dans la liste ouverte, ce dernier ne doit pas être dans la liste fermée ou dans la liste ouverte avec un coût plus faible. Le rôle de la liste fermée est d'éviter de rouvrir des noeuds déjà visités et donc de tomber dans une boucle sans fin et aussi de garder en mémoire le chemin vers l'état final. L'algorithme s'exécute jusqu'à ce que la liste ouverte soit vide, alors le chemin vers la solution est reconstruit à partir de la liste fermée. Voici l'algorithme A* en pseudo-code :

```
1: mettre le noeud de départ d dans la liste ouverte. f(d) = 0.
2: tant que la liste ouverte n'est pas vide faire
3:    rechercher dans la liste ouverte le noeud n ayant le plus faible f(n). Ce noeud est
      déplacé de la liste ouverte vers la liste fermée.
4:    pour tout Fils de n noté i faire
5:       mettre à jour le lien de parenté de i vers n.
6:       si i = but alors
7:          SORTIR
8:       fin si
9:       rechercher une occurrence de i dans la liste ouverte et dans la liste fermée.
10:      si aucune occurrence de i est trouvée, ni dans la liste ouverte ni dans liste fermée
         alors
11:         le mettre dans la liste ouverte
12:      fin si
13:      si une occurrence, notée k, est trouvée dans la liste ouverte et f(k) < f(i) alors
14:         ne pas rajouter i à la liste ouverte
15:      sinon
16:         supprimer k de la liste ouverte et rajouter i à la liste ouverte
17:      fin si
18:      si une occurrence, notée k, est trouvée dans la liste fermée et f(k) < f(i) alors
19:         ne pas rajouter i à la liste ouverte
20:      sinon
21:         supprimer k de la liste fermée et rajouter i à la liste ouverte
22:      fin si
23:   fin pour
24: fin tant que
```

Figure 2.9 – Pseudocode de l'algorithme A*

2.3.2.1 Application aux jeux vidéo

Comme dit précédemment, la recherche de chemins est utilisée dans les jeux vidéo pour déplacer les joueurs sur le terrain. Pour simuler, les endroits ou un agent peut se déplacer et donc exécuter l'algorithme A*, on utilise des grilles, des triangles, des graphes ou des «navigation mesh» [15]. Dans une grille, chaque cellule représente une position qu'un agent peut occuper. Certaines cellules peuvent être occupées par un obstacle les

rendant inaccessibles au déplacement des agents. Dans une grille, on peut seulement se déplacer à la cellule voisine et chaque déplacement à un coût unitaire. Les grilles se prêtent bien aux environnements rectangulaires comme des immeubles et d'autres milieux intérieurs. Lorsqu'on utilise de grands espaces non symétriques comme des environnements extérieurs, il vaut mieux utiliser des triangles. On appelle triangularisation de l'espace lorsqu'on divise l'espace en triangle. Les triangles n'ont pas tous la même forme et la même taille. En utilisant des triangles variés, on parvient à mieux représenter des terrains accidentés comme des flancs de montagnes. Tout comme avec les grilles, on peut se déplacer seulement à un triangle voisin. Le coût de déplacement d'un triangle à un autre est la distance entre le point de départ à l'intérieur du triangle et le côté commun au deux triangles. La triangularisation est un peu plus complexe en terme de nombre d'opérations que la recherche sur une grille, mais puisqu'un triangle a trois côtés et un carré quatre, il y a moins de noeuds à calculer. Un graphe fonctionne un peu de la même façon, on navigue d'un sommet à un autre en passant par les sommets. Le coût de déplacement de passer d'un sommet à un autre est variable et fixé au départ. Un «navigation mesh» est comme un graphe, mais les arêtes sont représentées par des volumes. Les agents se déplacent à l'intérieur du volume et ont donc une plus grande flexibilité de déplacement. Si un obstacle a un volume plus petit que celui du «navigation mesh» alors l'agent aura assez de place pour le contourner à l'intérieur du «navigation mesh». On pourrait, par exemple, représenter deux pièces reliées par une porte à l'aide de deux grilles, une pour chaque pièce et relier ces pièces par un graphe avec un sommet où il y

a la porte.

La plus grande difficulté de la recherche de chemins en IA est lorsqu'il y a des obstacles dynamiques. Dans un jeu où plusieurs agents se déplacent en même temps, il serait souhaitable que les agents n'entrent pas en collision, mais plutôt qu'ils coordonnent leur mouvement en prenant compte des mouvements des autres agents. Contrairement à des obstacles statiques où la recherche de chemin peut s'effectuer une seule fois avant le déplacement, les obstacles dynamiques demandent un recalcul continuel durant le déplacement. Concrètement, dans notre exemple précédent, si un agent devait traverser une pièce en même temps qu'un autre, un des deux devrait attendre que son compatriote soit traversé ou bien trouve un tout autre chemin pour traverser. À petite échelle, ce problème n'est peut-être pas d'une très grande importance, mais lorsque l'on crée des jeux de grande envergure avec plusieurs centaines d'agents se déplaçant en même temps, comme dans les jeux de stratégies en temps réel (STR), il faut penser à des stratégies de déplacement comme les mouvements par petit groupe. Pendant qu'un groupe se déplace, un autre peut les protéger des attaques ennemies.

2.3.3 Système multi-agents

Comme formuler à la section 2.3.1, généralement en jeux vidéo, les agents sont représentés par des personnages. Dans la littérature et dans ce mémoire, nous considérons qu'ils sont autonomes et produisent des actions sur leur environnement basé sur ce qu'ils perçoivent des autres agents ou de leur environnement.

Un système multi-agents (SMA) est composé de plusieurs agents qui interagissent entre eux puisqu'un agent peut ne pas tout connaître sur son environnement à tout moment. Les SMA peuvent être coopératifs lorsque les agents du système s'aident à atteindre leurs buts ou compétitifs lorsque les agents à l'intérieur du système sont en compétition pour les ressources. Il y a aussi des SMA où les agents sont en coopération à l'intérieur du système, mais en compétition avec les agents d'un autre SMA. Chaque agent est autonome, mais a obligatoirement besoin de l'aide des autres agents du système pour atteindre son but. Les SMA appliqués aux jeux vidéo font l'objet de contraintes supplémentaires :

- Dynamisme : La situation des agents change continuellement. Les agents se déplacent, interagissent avec d'autres agents et l'environnement change suite aux actions des agents. Ces derniers doivent constamment réviser leur situation afin de choisir la meilleure action à un moment donné en fonction de leurs capacités.

- Organisation du système : Dans un SMA les agents ne sont pas nécessairement tous égaux. Il peut y avoir une division des tâches ou bien une hiérarchie au sein de l'équipe faisant en sorte que pour un problème donné, ce ne sont pas tous les agents qui ont la capacité de le résoudre. C'est généralement à la phase du design que l'on distribue les rôles aux agents du système, mais ils peuvent aussi changer dynamiquement au cours du jeu [9].

- Communication : La qualité et la quantité des informations transmises entre les

agents d'un SMA a un impact sur les actions de ceux-ci. En général, plus une équipe échange de message plus sa performance augmente sauf dans le cas d'une surcharge [8]. Les agents ont donc intérêt à s'échanger des messages pertinents pour arriver à leur objectif. On peut aussi mentionner la sécurité des messages échangés. Dans le cas de SMA compétitif, les stratégies pour vaincre l'adversaire doivent rester secrètes afin de ne pas se faire piéger.

- Coordination : Dans un SMA la coordination est très importante. Les agents doivent être capables de s'entraider dans le cas de système coopératif. Si un agent meurt ou échoue sa tâche, un autre agent doit être capable de prendre la relève.

2.3.3.1 Application au jeux vidéo

Les SMA sont souvent utilisés pour représenter des groupes ou des équipes. Dans les jeux de sports, on peut représenter les deux équipes par deux SMA comprenant plusieurs agents, les joueurs. Dans les jeux de stratégie en temps réel, on utilise les systèmes multi-agents pour représenter des armées. Chaque armée peut être vue comme plusieurs soldats autonomes communiquant entre eux. Plusieurs autres types de jeux vidéo peuvent être vus comme des systèmes multi-agents lorsqu'on peut imaginer un jeu avec plusieurs entités autonomes. Ces entités vont communiquer, coopérer et négocier en mesure d'atteindre leurs objectifs.

2.4 Résumé

De nos jours, les jeux vidéo sont de complexes applications dus au fait que l'industrie demande des produits de qualités de haut niveau. Le développement de jeux vidéo demande des compétences dans plusieurs domaines comme le graphisme, les communications réseau et l'intelligence artificielle. En ce qui a trait à l'intelligence artificielle, les joueurs demandent des comportements de plus en plus crédibles et sophistiqués, mais les résultats sont parfois insatisfaisants dû aux contraintes de l'industrie comme le temps et l'argent. L'intelligence artificielle appliquée aux jeux vidéo peut être vue comme un système très complexe. Plusieurs champs de l'intelligence artificielle peuvent être appliqués aux jeux vidéo et plusieurs de ces techniques furent développées bien avant l'arrivée des jeux. Trois de ces concepts seront utilisés dans ce mémoire soit le concept d'agent, la recherche de chemins et les systèmes multi-agents. Ces trois concepts sont intimement liés et comme nous le verrons dans les prochains chapitres, ils rendent le comportement des joueurs non humains beaucoup plus réalistes et l'expérience de jeu beaucoup plus divertissant.

CHAPITRE 3

CONCEPTION

3.1 Version de base

Le jeu développé dans ce mémoire est une version modifiée du jeu « Snake ». Il s'agit d'un jeu d'arcade inventé à la fin des années 70 [6]. Le joueur contrôle un serpent qui glisse sur un plan bordé par des murs. Le but du jeu est de manger le plus de points de nourriture possible sans mourir. Le serpent meurt lorsqu'il entre en contact avec les murs ou son propre corps. Chaque fois que le serpent mange un point de nourriture, son corps s'allonge ce qui fait augmenter la difficulté au fil du temps. La version originale ne comportait qu'un seul niveau, mais il existe aussi des versions avec plusieurs niveaux où l'on peut trouver des murs à l'intérieur de l'aire.

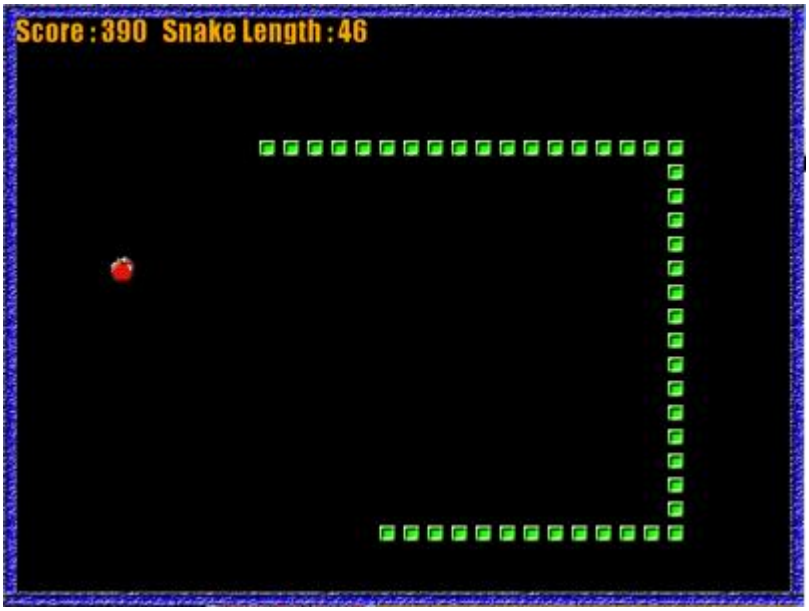

Figure 3.1 – Version de base du jeu développé

Cette version ne comporte qu'un seul agent (le serpent) contrôlé par le joueur qui cherche à accumuler le plus de ressources possible. Cependant, avec quelques modifications, il peut facilement être transformé en version multi-agents.

3.2 Version modifiée

Plusieurs modifications ont été faites à la version de base afin de créer une version multi-agents.

- Il y a au maximum quatre serpents en même temps dans l'aire de jeu. Le joueur en contrôle un seul, les autres sont contrôlés par l'IA du jeu. Le joueur doit donc, en plus d'éviter les collisions avec les murs et son propre corps, éviter les collisions avec les autres serpents.

- Les serpents sont d'une longueur fixe. Lorsqu'un serpent mange un point de nourriture, son corps ne s'allonge pas. Cependant, les points de nourritures ont une certaine valeur. Manger un point de nourriture ajoute sa valeur au score total du joueur.

- Puisqu'il y a plus d'un serpent, il y a aussi plusieurs points de nourriture. Les points de nourriture ont différentes valeurs représentées par des couleurs différentes. Ils apparaissent à n'importe quel moment dans l'aire de jeu.

- Afin que l'utilisateur ne se lasse pas trop vite, les collisions avec les murs ou les autres serpents n'arrêtent pas le jeu, mais font perdre des points.

- Les parties ont une durée prédéterminée.

3.2.1 But du jeu

Le but du jeu est de terminer la partie avec le plus haut score parmi tous les serpents. On doit donc manger le plus de points de nourritures sans entrer en collision.

Dans la prochaine section, les différents aspects du jeu seront développés plus en détail.

3.3 Environnement

3.3.1 Propriétés

L'environnement constitue l'espace de jeu. Il s'agit d'un environnement entièrement observable. Les serpents ont accès en tout temps à toute l'information pertinente sur l'environnement. Concrètement, il ont accès en tout temps à la position des points de nourriture, la position des autres serpents, l'emplacement des murs et la position du joueur.

L'environnement est déterministe. Son état dépend des actions des agents. Lorsqu'un agent mange un point de nourriture, il disparaît et un autre point apparaîtra ailleurs. Si aucun agent ne mange un point de nourriture alors l'environnement ne changera pas

Il s'agit d'un environnement non épisodique, son état dépend strictement des actions des agents. De plus, la position des points de nourritures est totalement aléatoire.

L'environnement est dynamique, comme dit précédemment, un point de nourriture peut apparaître à tout moment et la position des autres agents change aussi en tout temps.

Il est aussi discret puisque les serpents peuvent se diriger vers autant de directions différentes qu'il y a de points de nourriture et ont un nombre limité de stratégies pouvant être utilisées.

3.3.2 Points de nourriture

Les points de nourriture apparaissent aléatoirement dans l'aire de jeu. Il y a trois types de points de nourriture dans le jeu. La nourriture or vaut cinq points, celle argentée vaut trois points et la bronze un point. Lorsqu'un point de nourriture est consommé par un agent, sa valeur est ajoutée au score de l'agent et il disparaît. Il ne peut donc plus être mangé par un autre agent.

3.3.3 Collisions

Les agents doivent éviter autant que possible les collisions. Entrer en collision avec son propre corps, un autre serpent ou les murs qui bordent l'espace de jeu fait perdre 3 points. C'est la tête de l'agent qui détecte les collisions. Lorsque celle-ci entre en collision, l'agent est retiré du jeu pour une courte période de temps et reprend de sa position de départ. Si deux agents entrent en collision tête première alors les deux seront pénalisés.

3.4 Agent

Dans ce jeu, les agents sont les serpents. Ils se déplacent de façon autonome et puisqu'il n'y a pas d'obstacles dans le jeu sauf eux-même, ils savent où se trouvent les points de nourritures et les autres serpents. Un des serpents est contrôlé par le joueur, les trois autres sont des agents autonomes et formeront le système multi-agents qui compétitionnera contre l'utilisateur.

Il s'agit d'un système multi-agents coopératifs, les trois serpents formant le système s'entraident afin de vaincre l'utilisateur. Pour vaincre l'utilisateur, il suffit que le joueur humain ne finisse pas la partie avec le plus de points donc que seulement un des trois agents du système termine avec plus de points que l'utilisateur. Dans la section 3.4.1, je discuterai des stratégies utilisées pour ramasser des points ou en faire perdre au joueur humain.

3.4.1 Stratégies

Voici les stratégies de base que les agents utilisent :

1. Aller chercher le point de nourriture le plus près. Le serpent va chercher le point de nourriture le plus près de lui. Les points de nourritures sont parcourus selon leur distance avec le serpent.

2. Aller chercher le point de nourriture avec la plus grande valeur. La distance est calculée comme pour la stratégie précédente, mais cette fois-ci les points de nourritures sont parcourus selon leur valeur donc les points or en premier, ensuite, les points argent et finalement les points bronze. Si deux points ou plus de même valeur se retrouvent sur la surface de jeu, le serpent choisira celui qui est apparu en premier.

3. Tenter de voler les points de nourriture que le joueur humain convoite en se déplaçant vers le point de nourriture le plus près du joueur humain.

4. Éviter les autres serpents en s'isolant dans une zone libre. Puisque l'aire de jeu est un carré, on peut la diviser en quatre parties égales selon l'axe X et Y. Une zone libre est un quadrant. Puisqu'il y a quatre serpents et quatre quadrants, il y a toujours un quadrant libre où un serpent peut se diriger.

5. Sacrifier une partie de ses points en bloquant le joueur humain.

6. Tous les serpents formant le système multi-agents tentent de bloquer le joueur humain. Cette stratégie est semblable à la précédente, mais elle implique tous les agents et une coopération de ceux-ci.

3.4.2 Modélisation

Puisque les agents possèdent un but soit de terminer la partie avec le plus de points possible, ils sont de type délibératif. Les actions qu'ils performent sont toujours en fonction de ce but. Ils conservent aussi une trace des états antérieurs en gardant en mémoire leur destination qui peut être un point de nourriture ou le serpent humain. Ils gardent aussi en mémoire leur score total ainsi que le score des autres serpents afin de réaliser certaines stratégies plus avancées.

Les agents peuvent performer plusieurs stratégies différentes. Les stratégies sont prédéfinies et étant donné que les agents ne sont pas apprenants, le processus de sélection de la stratégie est semi-aléatoire. Chaque fois que le serpent se rend à sa destination ou entre en collision, une nouvelle stratégie est choisie en fonction de l'état de la partie. Certaines stratégies seront écartées à cause du classement ou de la position de l'agent,

et le serpent choisira aléatoirement une stratégie parmi celles restantes. Cette stratégie déterminera la nouvelle destination. Ce cycle se poursuit tout au long de la partie. Par exemple, un agent très loin du joueur humain ne pourra pas utiliser la stratégie qui tente d'aller voler les points de nourritures se situant près du joueur humain, de même, le serpent qui possède le plus de points dans le système multi-agents ne tentera pas de perdre son avance en bloquant le joueur humain

Voici un diagramme du système qui représente l'influence de l'environnement ainsi que de la communication inter-agents sur le comportement de l'agent. Ce dernier influence l'environnement lorsqu'il mange un point de nourriture et peut aussi influencer le joueur humain s'il décide de bloquer ce dernier.

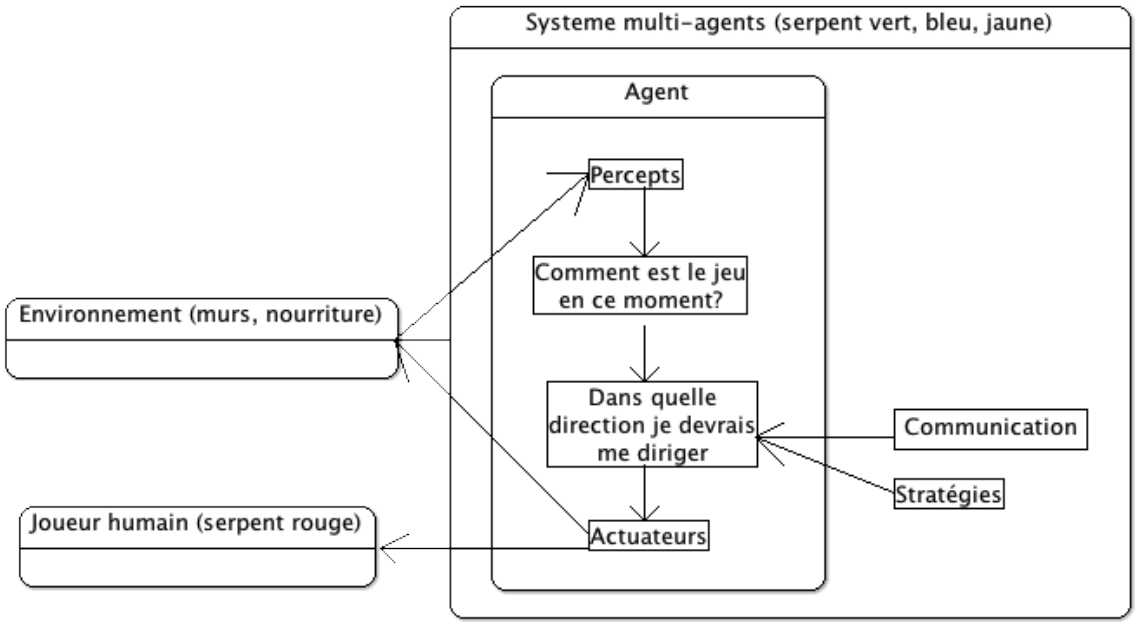

Figure 3.2 – Schéma d'un agent du jeu

Concrètement, l'arbre de décision représentant le comportement d'un agent du jeu

est illustré à la figure 3.3. La racine de l'arbre est constituée d'un séquenceur. Ce séquenceur exécutera tous ces fils de gauche à droite jusqu'à ce qu'un de ceux-ci échoue. Le premier fils vérifie que l'agent est arrivé à destination et qu'il n'est donc pas en train de se déplacer. Si c'est le cas, l'agent exécutera les prochaines actions.

Ensuite, l'agent utilisera un sélecteur pour fixer sa destination. Ce sélecteur choisira aléatoirement un de ses fils jusqu'à ce qu'un de ceux-ci réussisse. Chaque fils représente une des stratégies énoncées à la section 3.4.1. Les deux dernières stratégies ont besoin de satisfaire une condition supplémentaire. Pour aller voler un point de nourriture au joueur humain, l'agent doit être suffisamment près de ce dernier et pour bloquer le joueur humain, l'agent ne doit pas être le premier au classement.

Finalement, l'agent se déplace vers la destination trouvée. Cet arbre est exécuté en boucle tout au long du jeu.

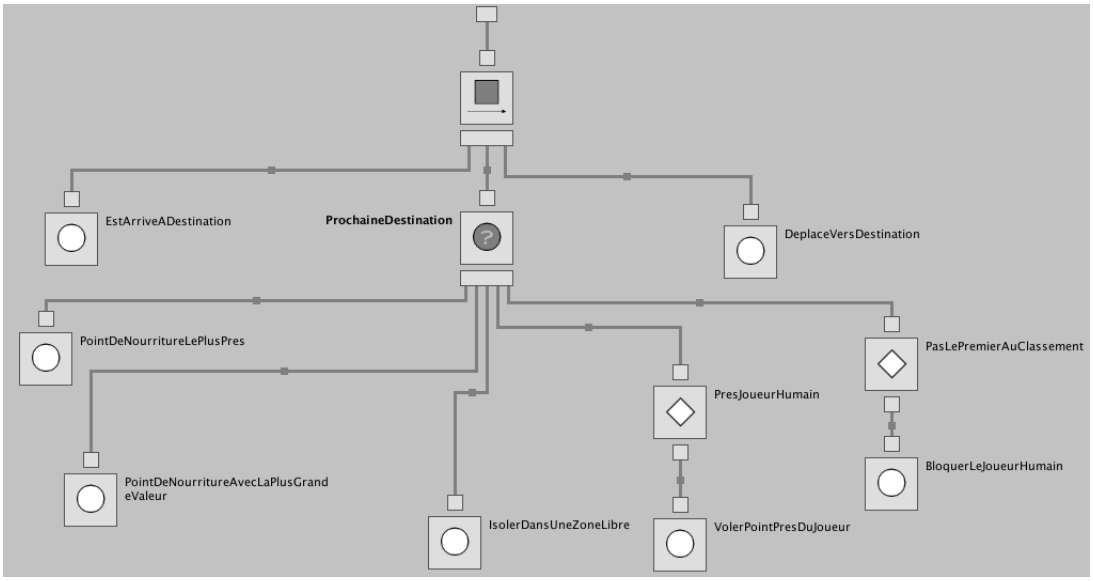

Figure 3.3 – Arbre de décisions d'un agent du jeu

3.4.3 Recherche de chemins

Pour se déplacer, les agents utilisent un algorithme de type A* [19]. Au moment de réaliser le jeu, il n'existait pas de fonctions sous *Unity* pour le faire. Avec la permission de l'auteur, les librairies « SimplePath » disponibles dans le marché de *Unity* [11] ont été utilisées. Dans le jeu, l'heuristique utilisée est la somme des distances selon l'axe x et l'axe y. Une grille invisible couvrant l'espace de jeu est située sous les serpents dans le plan XY. Tous les serpents se situant sur la grille occupent certaines cases. Ces cases ne peuvent donc pas être utilisées dans la planification de chemins. Une fois que la destination a été établie selon l'arbre de décision, on calcule sur quelle case ce point se trouve et l'on détermine le plus court chemin en appliquant récursivement la fonction d'évaluation aux cases voisines de la case de départ. Les cases sont carrées et de dimension égale au diamètre de la tête du serpent (la portion du serpent la plus large) afin que le serpent puisse passer par une case libre bordée par deux cases occupées.

3.5 Système multi-agents

Les agents ont tous le même but individuel soit « d'accumuler plus de points que le joueur », mais ils ont des conflits d'accès aux ressources. Il n'est pas impossible que deux serpents choisissent le même point de nourriture et donc que les agents à l'intérieur du système se gênent. Pour remédier à cette situation, il y a une collaboration entre les agents. Il s'agit donc d'un système multi-agents coopératif. Lorsqu'un agent se dirige vers un point de nourriture, il informe les autres agents de sa destination. En sachant,

où les autres agents du système se dirigent, un membre évitera de se diriger vers le même point de nourriture que ses coéquipiers. La coordination est dynamique, les agents envoient des messages contenant leur destination aux autres chaque fois qu'ils changent de destination. Il n'y a pas de négociations. Un agent reçoit la destination d'un autre agent et évite cette destination.

Une deuxième interaction existe entre les agents. Il s'agit de l'application de la cinquième et sixième stratégie de jeu mentionné dans la section 3.4.1. À tout moment, un agent peut, seul, tenter de bloquer le joueur humain ou peut envoyer aux autres agents un message demandant leur aide pour bloquer le joueur. Les trois serpents se dirigeront ainsi vers le joueur en le forçant à entrer en collision avec leurs corps. Encore une fois, il n'y a pas de négociations, lorsqu'un agent reçoit ce message, il délaisse sa destination pour se diriger vers l'utilisateur.

Étant donné qu'il y a une compétition pour les ressources entre le joueur et le système multi-agents, les agents doivent se regrouper sous forme de coalition pour atteindre leurs buts. Les agents vont collaborer pour coordonner leurs mouvements afin d'avoir de meilleures chances de finir la partie avec plus de points que le joueur.

CHAPITRE 4

IMPLÉMENTATION ET ÉVALUATION

4.1 Plateforme de développement

Le jeu pour cette maîtrise a été développé avec *Unity* de *Unity Technologies*. *Unity* est un moteur de jeu 2d et 3d disponible pour Mac OS X et Windows. Il peut être utilisé pour créer des jeux, des simulateurs, des animations interactives, etc. Une version gratuite permet de créer des versions mac, pc et web. On peut aussi acheter des licences pour *IOS (Iphone et Ipad), Android, Windows phone, Xbox 360, Playstation 3* et pour la *Wii*. Puisque *Unity* supporte beaucoup de plateformes et s'occupe de générer les projets pour la plateforme choisie, il permet de gagner beaucoup de temps à la période de déploiement.

Unity n'est pas un logiciel pour créer des animations ou des vidéos. Tout le contenu artistique doit être créé avec d'autres logiciels, comme *Blender, Maya, Photoshop, After Effects*, etc. *Unity* supporte les fichiers directement créer avec ces logiciels donc l'importation dans *Unity* se fait en un simple clique. *Unity* est relativement facile à apprendre, car il inclut beaucoup d'options et de fonctions prédéfinies. La physique, les collisions, les «shaders» et les effets de lumières sont quelques-unes des nombreuses fonctionnalités incluses et gérées par *Unity*. Avec la hausse en popularité des jeux pour consoles mobiles, *Unity* serait le choix le plus populaire selon le site web Gamasutra

[1]. Quelques-uns des titres les plus populaires inclus : Temple Run, Dead Trigger, Bad Piggies et Shadowgun.

La programmation sous *Unity* se fait avec *MonoDevelop*, un logiciel «open source» pour les langages .Net inclus dans le SDK de *Unity*. En synchronisant notre projet avec *MonoDevelop*, on obtient une aide s'ajustant à l'avancement de notre projet permettant de déboguer en temps réel dans le simulateur de *Unity* ou sur une vraie console.

4.2 Jeu développé

Le jeu développé est un jeu en deux dimensions avec une vue de haut vers le bas aussi appelé «Top-down view» en anglais. Il est à noter que les objets et personnages sont eux en trois dimensions. Les murs de couleur gris bordent l'espace de jeu et empêchent les serpents de sortir. Le sol est de couleur noire afin de pouvoir mieux voir les serpents de couleur rouge, jaune, bleue et verte. Finalement les points de nourriture ellipsoïdaux sont de couleur or, argent et bronze. Toutes ces formes et couleurs sont incluses comme forme de base dans le logiciel *Unity* donc aucun autre outil de conception 3D n'a été nécessaire pour concevoir le jeu.

Voici un exemple d'un match en cours.

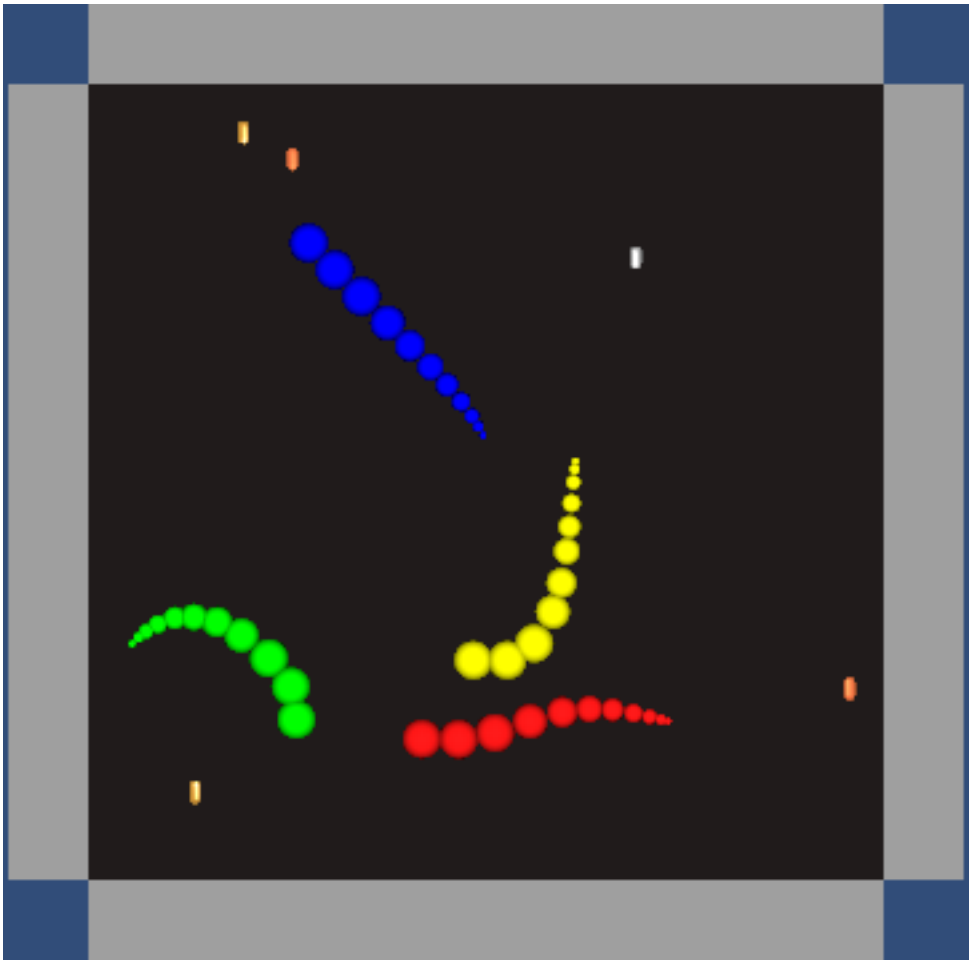

Figure 4.1 – Version multi-agents développée pour ce mémoire

4.2.1 Agents

Les serpents sont formés de onze sphères de diamètre décroissant. Chaque serpent est de couleur différente afin de les différencier. Les agents ainsi que le joueur humain qui contrôle un serpent possèdent la classe de base nommée «Snake» qui garde en mémoire le score du serpent, sa position de départ, les sphères formant son corps et qui s'occupe de gérer les collisions avec les autres serpents ou les murs. Pour gérer les collisions, chaque

mur et chaque sphère d'un serpent possèdent ce qu'on appelle en anglais un «collider» qui est généralement une boîte, mais peut aussi être un autre volume délimitant l'espace occupé par le modèle 3D. Lorsque deux «collider» entrent en contact, il y a collision. Plusieurs formes de «collider» sont déjà incluses dans *Unity*, ainsi que la fonction qui est appelée lorsqu'un «collider» entre en contact avec un autre.

La fonction nommée «OnTriggerEnter» est la fonction utilisée lorsque le serpent entre en collision. Le paramètre de la fonction est l'objet dans lequel est entré en collision le serpent. S'il s'agit d'un mur ou d'un autre serpent, on enlève 3 points et on appelle la fonction «spawn» qui s'occupe de replacer le serpent à sa position de départ. Un cas particulier est lorsque deux serpents entrent tête première l'un dans l'autre. À ce moment, les deux serpents sont pénalisés et perdent des points. Si au contraire il s'agit d'un point de nourriture, on ajoute au score du serpent la valeur du point de nourriture et on appelle «spawn» sur le point de nourriture qui se charge de le changer de position et de valeur aléatoirement le point de nourriture.

La classe «Enemy» hérite de la classe «Snake» et désigne les agents du jeu. C'est dans cette classe qu'est contenu l'arbre de décision qui gère le comportement des agents. Elle garde en mémoire, les destinations des coéquipiers, sa propre destination ainsi que la position du joueur humain. C'est cette classe qui s'occupe de déterminer la prochaine destination en fonction de la stratégie choisie. Elle se charge aussi d'appeler le module de recherche de chemins pour le déplacement.

4.2.1.1 Stratégies

Tel que mentionné à la section 3.4.1 , il y a 6 actions possibles pour un agent. La prochaine capture d'écran illustre quelques stratégies. Ici, le serpent vert se dirige vers le point de nourriture le plus près, le serpent bleu se dirige vers un des points de nourriture ayant la plus grande valeur et le serpent jaune se dirige vers une zone libre (le quatrième quadrant). Le serpent rouge est contrôlé par le joueur.

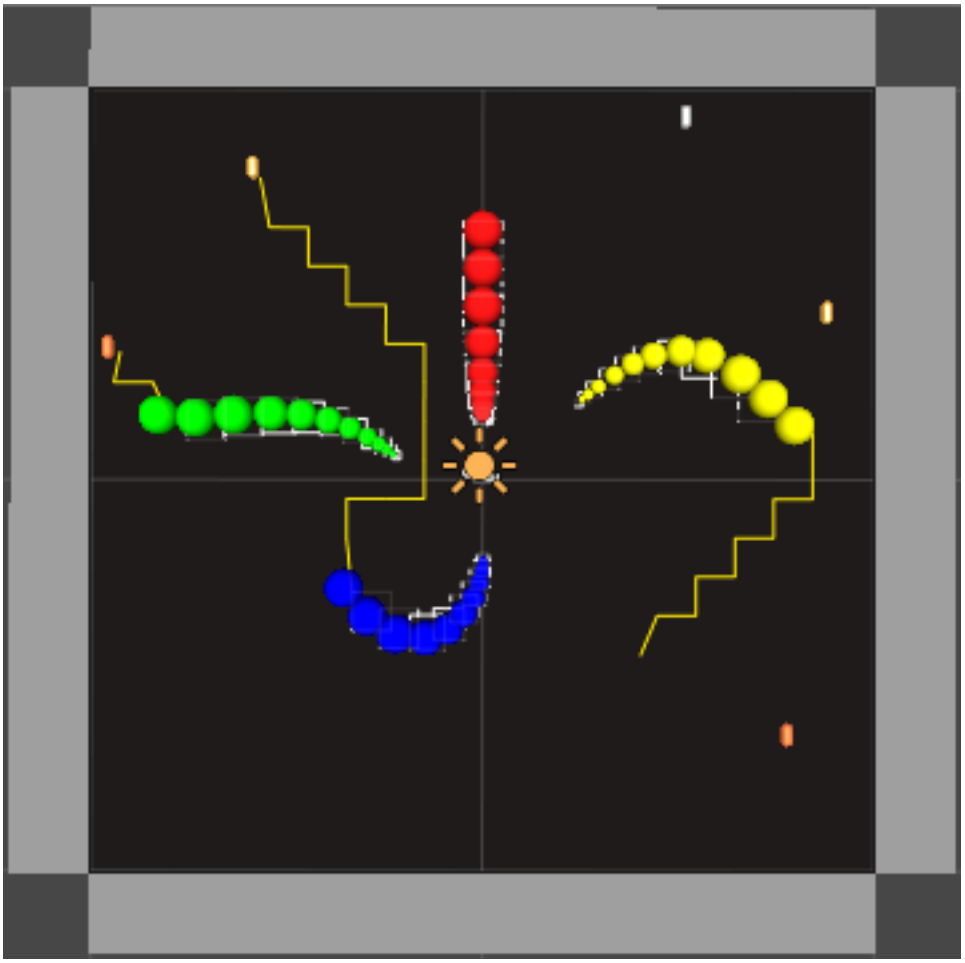

Figure 4.2 – Différentes stratégies utilisées par les agents du système multi-agents

L'arbre de décisions se charge de choisir la prochaine destination de l'agent.

4.2.2 Recherche de chemins

Comme mentionnée à la section 3.4.3, une grille invisible est placée sous le serpent afin de déterminer quelles cases sont libres pour planifier le chemin à l'aide de l'algorithme A*. Pour chaque sphère d'un serpent, on détermine la ou les cases qui se trouvent vis-à-vis en dessous. Ces cases sont maintenant marquées comme étant non accessibles pour l'algorithme A*. On débute ensuite l'algorithme A* en partant de la case sous la tête du serpent puis on visite récursivement chaque case adjacente à la case en cours en choisissant celle qui à la moins grande distance jusqu'à la case but. La distance entre deux cases est déterminée en additionnant le nombre de case les séparant selon l'axe horizontal et l'axe vertical. Voici un exemple illustrant la recherche de chemins, les cases obstruées sont rouges dans la figure 4.3.

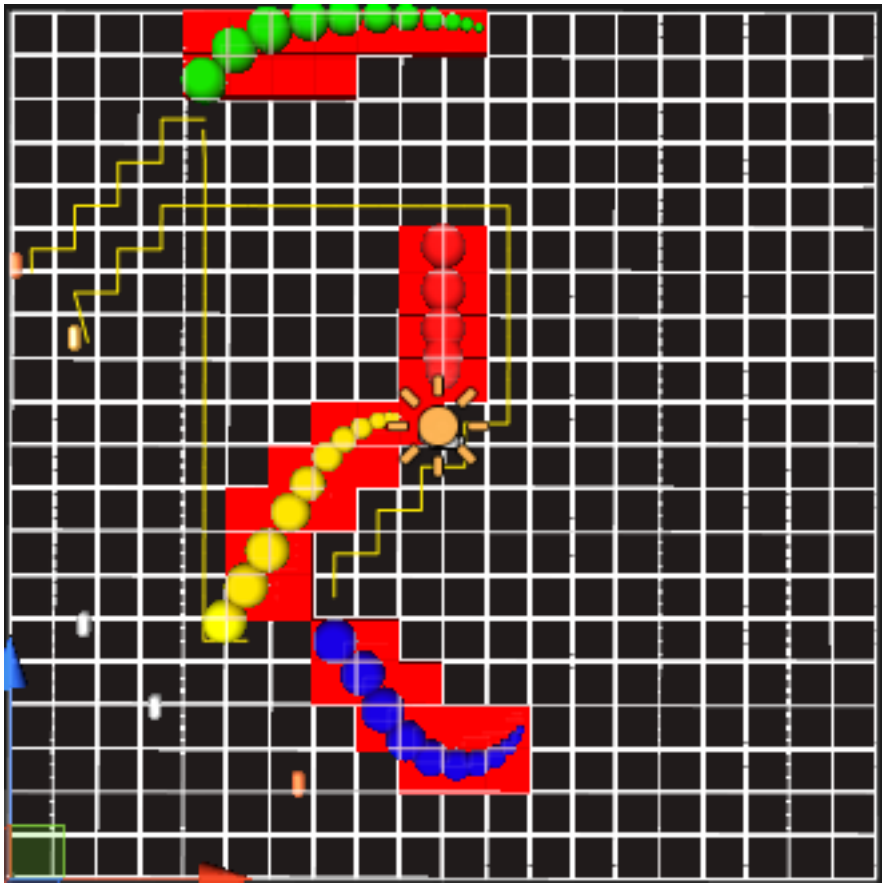

Figure 4.3 – Capture d'écran illustrant la recherche de chemins

La fonction débute par prendre un noeud de la liste ouverte qui contient les noeuds non visités. Si ce noeud est aussi le noeud final, alors on peut construire la solution et sortir de la fonction. Si ce n'est pas le cas, on calcule le coût de chaque noeud adjacent au noeud courant. Si le noeud adjacent n'a pas déjà été évalué, on enregistre son coût et son noeud parent qui est le noeud courant afin de pouvoir reconstruire le chemin. Si le noeud a déjà été calculé, on compare son coût au coût calculé auparavant. Si le coût actuel est moindre que celui déjà calculé, cela veut dire que nous avons trouvé un chemin

moins coûteux pour arriver à ce noeud. On actualise alors le noeud en enregistrant son nouveau noeud parent et son coût.

4.2.3 Système multi-agents

Tel que mentionné au chapitre précédent, les agents forment un système multi-agents coopératif. Ils s'entraident afin de vaincre le joueur humain de plusieurs façons :

1. Il ne se dirige pas vers un point de nourriture convoité par un autre agent. La figure 4.4 démontre cette coopération. Le serpent vert utilise la stratégie du point de nourriture le plus près. Il devrait donc se diriger vers le point de nourriture or en haut à droite. Cependant, le serpent bleu se dirige déjà vers ce point de nourriture, le serpent vert ira alors au deuxième point de nourriture le plus près évitant ainsi une collision.

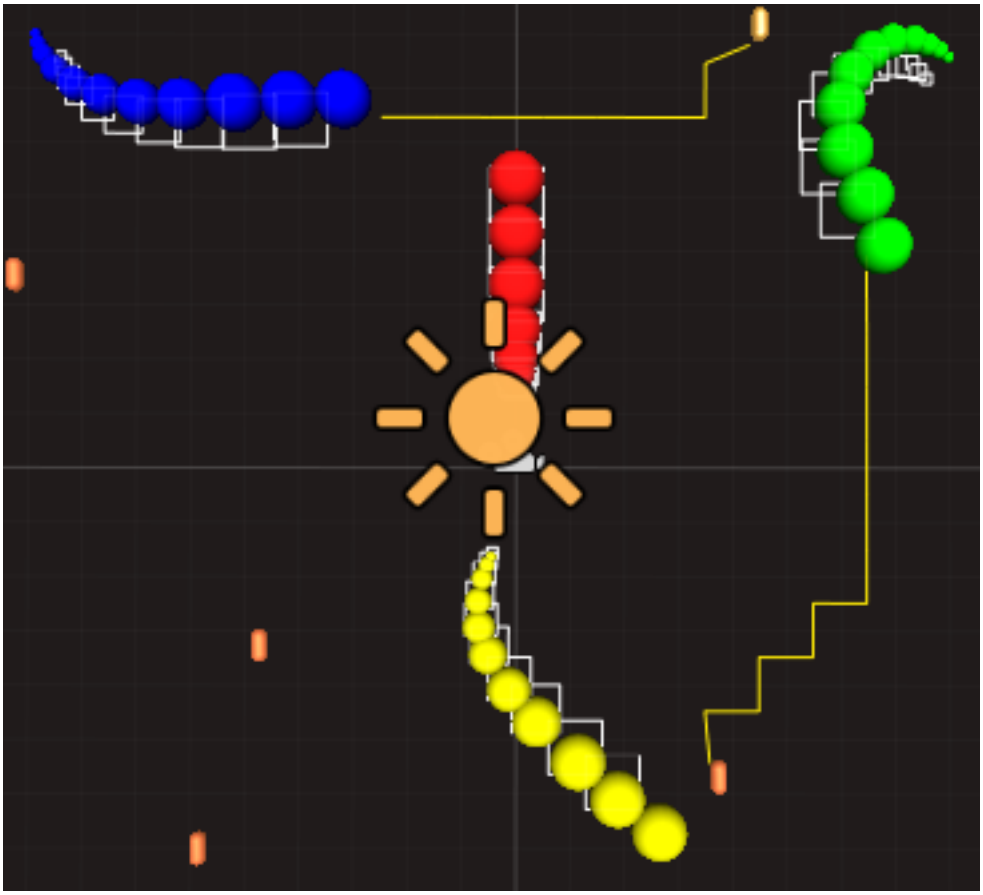

Figure 4.4 – Capture d'écran démontrant la coopération entre agents

2. L'agent qui possède le moins de points peut sacrifier une partie de ses points pour tenter de bloquer le joueur humain. Cette option est illustrée à la figure 3.3. En tentant d'entrer en collision avec le joueur humain et de le déconcentrer. On aide l'agent premier au classement à rattraper le joueur humain ou bien à conserver son avance.

3. Finalement, en dernier recours, les agents peuvent tous simultanément unir leurs forces et effectuer la sixième stratégie présentée à la section 3.4.1 qui est de blo-

quer le joueur humain. Cette fois-ci, tous les serpents, en s'évitant les uns les autres, tenteront d'entrer en collision avec le joueur humain pour lui faire perdre ses points. Un exemple de cette stratégie est présenté dans la figure 4.5. Ici, le serpent rouge est contrôlé par le joueur et les autres serpents formant le système multi-agents tentent de le bloquer en se déplaçant devant lui.

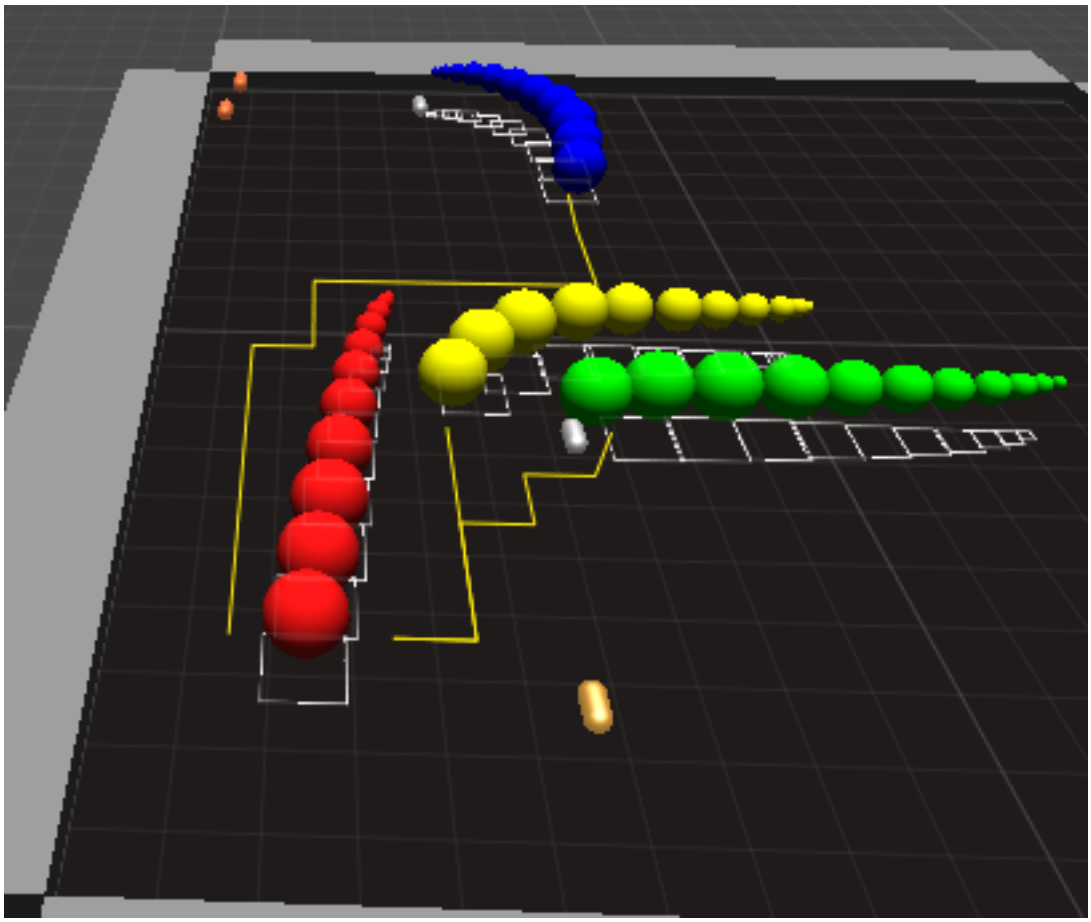

Figure 4.5 – Capture d'écran démontrant la coordination vers un but commun

4.3 Résultats et analyse

4.3.1 Agents

L'arbre de décision responsable du comportement des agents fut bien utile tout au long de ce mémoire. Un des avantages principaux des arbres de décision est que lorsque les fonctions élémentaires de l'arbre sont faites, on peut facilement changer la structure de l'arbre et ainsi tester plusieurs combinaisons de comportement. Parmi les six stratégies que les agents peuvent exécuter mentionné à la section 3.4.1, certaines se sont avérées plus efficace.

- La stratégie la plus efficace est celle où l'agent va chercher le point de nourriture le plus près de sa position actuelle. Bien qu'il n'est pas garanti que l'agent accumulera le maximum de points par déplacement, il amassera en moyenne deux points par déplacement puisque le point de nourriture le plus faible vaut un point et le plus élevé trois points. La valeur et la position des points sont définies au hasard. L'agent amassera donc de façon constante et rapide puisque les points sont près de lui, un petit nombre de points.

- Une autre stratégie intéressante est celle où l'agent se dirige vers le point de nourriture avec le plus de points. Cette stratégie est moins efficace que la première car elle implique plus de risques pour l'agent soit de se déplacer sur une plus grande distance. Puisque l'aire de jeu est relativement petite compte tenu du nombre d'agents, les collisions sont très fréquentes et donc cette stratégie mène bien sou-

vent à un échec dans le déplacement. Néanmoins, avec une bonne coordination entre les agents, cette stratégie peut être la plus payante.

- La troisième stratégie la plus payante est celle où l'agent tente de voler les points près du joueur humain. Cette stratégie n'est pas très efficace car il est très difficile pour l'agent de contourner le joueur humain et de lui voler ses points. Les mouvements de l'agent sont limités aux trois directions : avant, gauche et droite. Il doit aussi composer avec le comportement imprévisible de l'agent.

- La stratégie la moins payante est celle où l'agent s'isole dans un coin de l'aire de jeu afin de conserver son avance. Bien entendu, l'agent ne gagne pas de points, mais n'en perd pas non plus. Cette stratégie peut être utile lorsque combinée avec une autre comme la première. On accumule le plus de points afin de grimper en première place puis on utilise cette stratégie pour conserver la position de tête.

- Tout comme la stratégie précédente, la dernière stratégie, soit de bloquer le joueur humain, n'est pas très utile si elle est utilisée seule. On peut au mieux faire perdre tous les points à l'utilisateur, mais on doit aussi en amasser un peu pour le dépasser.

Ces résultats ont été observés après une vingtaine de parties. Le tableau 4.I illustre, pour chaque stratégie, la moyenne des points accumulés après un match. La première colonne inclue le joueur humain tandis que ce dernier n'a pas joué dans les simulations de la deuxième colonne. Les agents du mémoire peuvent exécuter toutes ces stratégies mais en les forçant à toujours exécuter la même, on peut comparer l'éfficacité de chaque

stratégie individuellement. Le score de chaque agent et du joueur humain ont été calculés en addititionnant tous les points de nourriture capturés selon leur valeur respective et en soustrayant les collisions qui font perdre chacune trois points. La présence du joueur humain rend le jeu un peu plus difficile pour les agents puisqu'il crée un obstacle de plus à éviter et peut potentiellement voler des points aux agents.

Stratégies	Score total avec joueur humain	Score total sans joueur humain
Point de nourriture le plus près	46.2	56.6
Point de nourriture avec le plus de points	39	48
Voler les points près du joueur humain	20	X
Isolement	-16.1	-14.4
Bloquer le joueur humain	-30.4	X
Joueur humain	37.3	X

Tableau 4.I – Résultats quantitatifs des stratégies

4.3.2 Recherche de chemins

L'algorithme A* fut parfait pour ce jeu. Puisque ce jeu se déroule très rapidement, les agents doivent réviser leurs chemins à courts intervalles, mais étant donné qu'il n'y avait que trois agents et que quatre cents cases sur lesquelles les agents peuvent se déplacer,

le recalcul n'est pas trop couteux en mémoire, mais ceci pourrait peut-être devenir un problème si le jeu incluait plus d'agents sur un terrain plus vaste. Il est plutôt facile de faire entrer en collision un agent dans lui-même ou un mur. Puisque chaque serpent ne peut qu'avancer sinon il serait en contact avec son propre corps, il y a toujours au maximum trois cases pour l'algorithme A*, soit la case devant le serpent ou celle de chaque côté. Lorsque le serpent tourne, il y a seulement deux cases disponibles pour le serpent, soit celle devant lui, et la case dans la direction opposée au virage. On peut alors tenter de suivre le serpent côte à côte du côté opposé au virage pour alors le restreindre à seulement aller vers l'avant. On peut alors forcer l'agent à se déplacer dans une direction non souhaitée.

4.3.3 Système multi-agents

L'intérêt de l'implémentation d'un système multi-agents dans un jeu vidéo est qu'il permet de répliquer les comportements humains de façon plus fidèle. Tout comme dans une équipe sportive, une armée ou bien tout autre groupe, généralement lorsque les membres s'entraident les performances sont meilleures. C'est ce qui a été observé avec le système multi-agents de ce mémoire.

Voici un petit rappel de la coopération à l'intérieur du système.

- Premierement, chaque agent ne se dirige pas vers un point de nourriture si ce dernier est déjà convoité par un autre agent. Ce faisant, les agents évitent d'entrer en collision au maximum, car ils ne se dirigent jamais vers la même destination.

- Deuxièmement, l'agent qui a accumulé le moins de points dans le système peut tenter de sacrifier une partie de ces points en tentant de bloquer le joueur humain. En bloquant le joueur humain, ce dernier sera à risque de faire un faux mouvement et entrer en collision soit avec lui-même, un mur ou l'agent. Si l'agent perd des points en effectuant cette manoeuvre, il l'aura fait au bénéfice de l'agent premier au classement pour que celui-ci augmente son avance face au joueur humain.

- Troisièmement, tous les agents peuvent tenter en même temps de bloquer le joueur humain. Ceci requiert un effort collectif de tous les agents du système.

Avec ces stratégies, chaque agent aurait de bien meilleures chances de remporter la partie que s'il n'y avait pas eu de système, car chacun bénéficie d'informations importantes sur les autres agents. L'agent peut ainsi coordonner ces mouvements en sachant la destination des autres agents afin d'éviter les collisions. Cela lui permet aussi de sauver du temps en ne déplaçant pas inutilement vers un point de nourriture déjà presque capturée par un de ces pairs. Finalement, si le joueur humain est premier au classement, le meilleur agent du système, donc le deuxième au classement, pourra compter sur ces coéquipiers pour tenter de faire perdre des points au joueur humain en forçant des collisions.

On peut quand même observer certaines limites à ce système.

- Premierement, même si chaque agent connaît la destination des autres agents du système, il ne connaît pas le chemin que ceux-ci vont emprunter et donc peut

quand même rester bloqué parce qu'un autre agent est passé tout juste devant lui. On pourrait envisager un système où l'agent, en plus de savoir la destination de ses coéquipiers, pourrait aussi savoir leur direction afin de ne pas passer devant ces derniers.

- Deuxièmement, les agents prennent en compte l'état actuel du jeu, mais n'ont pas de planification à long terme, ce qui peut créer des déplacements non optimaux. Par exemple, si un agent cherche le point de nourriture avec la plus grande valeur, mais que ce dernier est près d'un autre agent qui lui est en train de se déplacer vers un autre point de nourriture plus loin, il se dirigera tout de même vers ce point. Si chaque serpent pouvait prévoir sa prochaine destination, le premier serpent aurait pu trouver un autre point de nourriture potentiellement plus près de lui en sachant que le deuxième serpent aurait pris le point de nourriture.

4.3.4 Comparaison

Comme on peut le constater, la version multi-agents du jeu est très différente et beaucoup plus complexe que la version sans agents. Comme je l'ai mentionné dans les chapitres précédents, la version multi-agents amène plus de défis pour le joueur humain puisqu'il doit maintenant compétionner pour les ressources. Il doit donc non seulement finir la partie, mais en plus finir la partie avec plus de points que ses adversaires ; les agents. Le jeu demande une plus grande rapidité du temps d'exécution car les parties sont minutées et le joueur humain doit accumuler le plus de points tout en évitant les agents.

La version multi-agents demande aussi un peu plus de stratégies puisqu'on doit tenter de prévoir à l'avance quelle sera notre prochaine destination. Ceci demande une évaluation globale de l'état du jeu. Le jeu est plus difficile pour le joueur humain puisqu'il doit faire face à ces nouvelles contraintes, mais plus intéressant puisqu'il doit compétionner contre d'autres adversaires.

Au niveau technique, la version multi-agents est plus compliquée à développer car elle comporte tout le code et les contraintes de la version multi-agents en plus de toute la mécanique reliée aux agents et au système. Ce mémoire est basé sur la version mono-agents pour en construire la version multi-agents. Même s'il s'agit d'un jeu simple, il a demandé beaucoup de programmation et d'ajustements afin d'être jouable pour le joueur humain.

Quant au type d'agents, on pourrait imaginer ce jeu avec un autre type d'agents. Les agents réactifs seraient trop simple pour ce jeu puisqu'il demande une connaissance des états antérieurs du jeu comme le chemin pour se rendre à une destination. Aussi, l'agent doit se souvenir de sa destination. Les agents auraient cependant pu être de type apprenants. Avec certaines règles simples, ils pourraient apprendre sur les meilleures stratégies à utilisé et surtout s'adapter au niveau et au style de jeu du joueur. Ceci demanderait par contre encore plus de travail puisque cet apprentissage serait fait en plus des composantes de base de l'agent déjà présentes dans le jeu.

CHAPITRE 5

CONCLUSION

Le jeu développé pour ce mémoire démontre bien l'utilisation et l'importance d'un système multi-agents dans un jeu vidéo. Dans le jeu développé pour ce mémoire, les probabilités pour un agent de gagner la partie sont bien plus grandes lorsqu'il fait partie du système qu'individuellement. Cette différence s'explique principalement par la co-opération entre les agents. En effet, les agents s'échangent des informations importantes sur leurs destinations respectives et sur le joueur humain leur permettant ainsi de coordonner leurs mouvements afin d'éviter les collisions et d'augmenter leur chance de faire perdre des points au joueur humain. Le déplacement de chaque agent est réalisé à l'aide de l'algorithme A* qui est très utilisé dans l'industrie du jeu vidéo. Le comportement de chaque agent est réalisé avec un «arbre de décisions» qui permet plus de flexibilité au niveau du code qu'une machine à états finis.

5.1 Défis rencontrés

Le système multi-agents développé amène beaucoup plus de compétition pour le joueur humain et permet aux agents d'avoir un comportement plus semblable à celui d'une équipe réelle. Ce système a cependant un coût, puisque le jeu est dynamique et que les agents sont constamment en déplacement, ils doivent sans cesse réviser leurs chemins afin d'éviter les collisions. Dans la version finale du jeu de ce mémoire, il n'y

a que trois agents et l'aire de jeu est un carré de vingt par vingt cellules. Le temps CPU pour calculer les chemins est donc très rapide. Ceci pourrait devenir un inconvénient si le nombre d'agents ou la dimension du terrain augmentait.

Un autre défi à surmonter lorsqu'on développe un système multi-agents dans un jeu vidéo est de bien balancer la force des adversaires, qui sont dans ce cas-ci les agents du système. On veut que les agents prennent les meilleures décisions, mais que ceux-ci ne soient pas parfait, c'est-à-dire que comme un joueur humain, ils peuvent faire des erreurs. Si le système est bien balancé, le joueur humain ne se sentira pas frustré parce que ses adversaires sont trop bons ou bien il ne se sera pas ennuyé parce que ses adversaires sont trop mauvais. Une bonne façon de répondre à ce problème est l'utilisation d'agents apprenants. Les agents apprenants peuvent s'adapter au style de jeu du joueur humain et aussi s'adapter à son niveau. Puisqu'un joueur devient généralement meilleur à un jeu à force de jouer, l'agent apprenant devrait aussi devenir meilleur au fil du temps. Dû au temps necéssaire pour développer des agents apprenants, ce mémoire focalise plus sur l'implantation du système dans un jeu. Cependant, les stratégies utilisées par les agents dans ce jeu sont des stratégies que des joueurs humains utiliseraient puisqu'elles seraient utilisées par des joueurs humains. Le défi n'est donc pas de programmer des fonctions simples modélisant quelques décisions qu'un joueur humain peut prendre, mais plutôt de simuler un comportement humain très complexe qui comporte aussi une partie d'imprévisibilité.

Plusieurs défis techniques se sont aussi présentés lors de la conception de ce jeu. Le

premier fut le choix du type d'agent à utiliser. Puisque la version mono-agent du jeu comportait un but précis soit de ramasser le plus de points avant de mourir, le même but à été utilisé dans la version multi-agents. C'est ce but qui guide les agents dans leurs déplacements. Si les agents avaient été purement réactifs, ils n'auraient pas pu accomplir les stratégies car elles exigent une connaissance de l'environnement et une connaissance de l'état précédent du jeu.

Un autre défi technique de ce jeu est la recherche de chemins. Il a d'abord fallut faire un choix sur le design de l'implémentation de la recherche de chemins. Puisque l'aire de jeu est un carré sans obstacles, une grille se prêtait bien à ce terrain. Il a fallut quelques tests avant de trouver la bonne grandeur pour chaque carré. Plus les carrés étaient petits, plus le mouvement des agents était précis. Par contre, si le carré est plus petit que l'agent, cela implique qu'il pourrait entrer en collision sur les cotés si un autre agent se trouve sur la cellule avoisiante. Dans la version finale du jeu, la longueur des cotés de chaque cellule est égal au diamètre de la tête du serpent, soit la partie la plus large de son corps et donc la plus petite dimension possible tout en pouvant contenir completement un agent.

Finalement, la communication entre les agents a aussi posé problème. La première version du jeu comporte un système de messagerie entre les agents. Ce système a été mis de coté dans les versions ultérieures afin de se concentrer plus sur les stratégies des agents. Puisque Unity n'offre pas un tel système, il a fallu le créer. Les messages comprenaient de simples règles de logique permettant aux agents d'exprimer à leurs coéquipiers, leur destinations et certaines stratégies afin de bloquer le joueur humain.

5.2 Améliorations

Puisque le jeu original a été développé il y a trente ans, il peut être difficilement comparé aux jeux vidéo modernes en terme de complexité algorithmique et d'utilisation de la mémoire. Cette différence entre le jeu développé et les jeux modernes rend une extrapolation des résultats difficile. Ce mémoire présente un aperçu du potentiel de l'utilisation d'un système multi-agents dans le développement d'un jeu vidéo et les conclusions tirées ici devront être validées dans le futur par le développement de jeux modernes utilisant un système multi-agents. Il serait maintenant intéressant de voir quel serait l'impact de l'implantation d'un système multi-agents dans un jeu de qualité commerciale afin de tester plus profondément les limites que pourrait imposer le système sur des jeux plus exigeants, tant en utilisation processeur qu'en utilisation mémoire. Le système multi-agents pourrait aussi être approfondi en augmentant la quantité de messages échangés entre les agents ainsi que la complexité de ceux-ci. Finalement, un moteur d'inférence pourrait être attaché aux agents pour que ceux-ci puissent communiquer avec un formalisme plus élaboré et ainsi inférer de nouveaux faits ou de nouvelles règles à partir de celles échangées.

BIBLIOGRAPHIE

[1] Mobile game developer survey leans heavily toward ios, unity, Mai 2012. URL http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity.php#.UQ7K2qWGtFu.

[2] Fabio Aiolli et Claudio Palazzi. Enhancing artificial intelligence in games by learning the opponent,Äôs playing style. Dans Paolo Ciancarini, Ryohei Nakatsu, Matthias Rauterberg et Marco Roccetti, éditeurs, *New Frontiers for Entertainment Computing*, volume 279 de *IFIP International Federation for Information Processing*, pages 1–10. Springer Boston, 2008. URL http://dx.doi.org/10.1007/978-0-387-09701-5_1.

[3] Tomas Akenine-Möller et Jacob Ström. Graphics for the masses : a hardware rasterization architecture for mobile phones. *ACM Trans. Graph.*, 22(3):801–808, juillet 2003. ISSN 0730-0301. URL http://doi.acm.org/10.1145/882262.882348.

[4] Entertainment Software Association. Essential facts about the computer and video games industry, 2012. URL http://www.theesa.com/facts/pdfs/ESA_EF_2012.pdf.

[5] Vadim Bulitko, Yngvi Björnsson, Nathan Sturtevant et Ramon Lawrence. Real-

time heuristic search for pathfinding in video games. Dans *Artificial Intelligence for Computer Games*. Springer, 2010. In press.

[6] R. DeMaria et J.L. Wilson. *High Score !, Second Edition*. Computer Games Series. McGraw-Hill, 2003. ISBN 9780072231724. URL http://books.google.ca/books?id=HJNvZLvpCEQC.

[7] Chris Fairclough, Michael Fagan, Brian Mac Namee et Pádraig Cunningham. Research directions for ai in computer games. Dans *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*, pages 333–344, 2001.

[8] Susan R. Fussell, Robert E. Kraut, F. Javier Lerch, William L. Scherlis, Matthew M. McNally et Jonathan J. Cadiz. Coordination, overload and team performance : effects of team communication strategies. Dans *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 275–284, New York, NY, USA, 1998. ACM. ISBN 1-58113-009-0. URL http://doi.acm.org/10.1145/289444.289502.

[9] In-Cheol Kim. Dynamic role assignment for multi-agent cooperation. Dans *Computer and Information Sciences ,Äì ISCIS 2006*, volume 4263 de *Lecture Notes in Computer Science*, pages 221–229. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-47242-1. URL http://dx.doi.org/10.1007/11902140_25.

[10] Thomas Klemensen et Wesley Iliff. Artificial intelligence in video games : A survey. University of Northern Iowa, 2010.

[11] Alex Kring. Simplepath. Unity asset store, Mai 2011. URL http://u3d.as/content/alex-kring/simple-path/1QM.

[12] Marie-Josée Legault. L'industrie du jeu vidÉo, un tÉmoin de la transformation contemporaine du travail hautement qualifiÉ. *Nouveaux cahiers du socialisme*, 7, 2011.

[13] Chong-U Lim, Robin Baumgarten et Simon Colton. Evolving behaviour trees for the commercial game defcon. Dans *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 100–110, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12238-8, 978-3-642-12238-5. URL http://dx.doi.org/10.1007/978-3-642-12239-2_11.

[14] Kathryn Merrick, Mary Lou Maher, Kathryn E. Merrick et Mary Lou Maher. Non-player characters in multiuser games. Dans *Motivated Reinforcement Learning*, pages 3–16. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-89187-1.

[15] Alexander Nareyek. Ai in computer games. *Queue*, 1(10):58–65, février 2004. ISSN 1542-7730. URL http://doi.acm.org/10.1145/971564.971593.

[16] C. Adam Overholtzer et Simon D. Levy. Evolving ai opponents in a first-person-shooter video game. Dans *Proceedings of the 20th national conference on Artificial intelligence - Volume 4*, AAAI'05, pages 1620–1621. AAAI Press, 2005.

ISBN 1-57735-236-x. URL http://dl.acm.org/citation.cfm?id=1619566.1619585.

[17] Amit Patel. Amit's a* pages, November 2012. URL http://theory.stanford.edu/~amitp/GameProgramming/.

[18] Alex "Sandy" Pentland. The new science of building great teams. *Harvard Business Review*, page 11, April 2012.

[19] Stuart Russell et Peter Norvig. *Artificial Intelligence : A Modern Approach*. Prentice Hall, third édition, décembre 2009. ISBN 0136042597. URL http://www.worldcat.org/isbn/0136042597.

[20] Jonathan Schaeffer. A gamut of games. *AI Magazine*, 22(3):29–46, 2001. URL http://dblp.uni-trier.de/db/journals/aim/aim22.html#Schaeffer01.

[21] Alexander Shoulson, Francisco M. Garcia, Matthew Jones, Robert Mead et Norman I. Badler. Parameterizing behavior trees. Dans *Proceedings of the 4th international conference on Motion in Games*, MIG'11, pages 144–155, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25089-7. URL http://dx.doi.org/10.1007/978-3-642-25090-3_13.

[22] Chek Tien Tan et Ho-lun Cheng. A combined tactical and strategic hierarchical learning framework in multi-agent games. Dans *Proceedings of the 2008 ACM*

SIGGRAPH symposium on Video games, Sandbox '08, pages 115–122, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-173-6. URL http://doi.acm.org/10.1145/1401843.1401865.

[23] Michael Woolridge et Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley &amp ; Sons, Inc., New York, NY, USA, 2001. ISBN 047149691X.