Simulation Comportementale dans les Environnements Virtuels

Pamela Carreño Medrano

Ecole Nationale d'Ingénierie de Brest

Institut de Rercherche en Informatique de Toulouse - Équipe Vortex

Master Informatique Spécialité Recherche en Informatique

Simulation Comportementale dans les Environnements Virtuels

Étudiante :
Pamela Carreño Medrano

Encadrants :
Dr. Jean-Pierre Jessel
Dr. Cedric Sanza

5 juin 2012

Table des matières

Table des figures

Liste des tableaux

Abstract

This document explores the use of Behavior Trees as a suitable technique for behavioral simulation of non-player characters, in the context of Serious Games programming. Our main objective is to propose a proper, extensible, reusable and simple behavioral method that would aid us in the generation of scripted adaptive autonomous agents, as well as in the successful transmision of Serious Game content. Here, we present the basis of such behavioral method and implementation based on the concept of coroutines and developped under the Unity3D Game Engine. Finally we introduce a simple use case based on a Serious Game concept that has as main objective to show the proper way for a passenger to take the subway.

Keywords : Serious Games, behavior trees, behavioral simulatiion, non-player characters, coroutines, autonomous agents.

Résumé

Ce document explore l'utilisation des arbres de comportement comme technique appropriée pour la simulation comportementale des personnages non-joueurs, dans le cadre de la programmation de jeux sérieux. Notre objectif principal est de proposer une méthode appropriée, extensible, réutilisable et simple permettant la génération d'agents autonomes adaptatifs scénarisés, ainsi que la transmission réussie de contenu sérieux. Ici, nous présentons la base de la méthode comportementale proposée, puis une implémentation basée sur le concept de coroutines et développé sous le moteur du jeu Unity3D. Finalement, nous présentons un cas d'utilisation simple basé sur un concept de jeu sérieux, qui a pour objectif principal de montrer la façon appropriée selon laquelle une passager doit prendre le métro.

Mots clé : jeux sérieux, arbres de comportement, simulation comportementale, personnages non joueurs, coroutines, agents autonomes.

Introduction

Ces dernières années, les **Serious Games** ou **Jeux Sérieux** ont commencé à devenir de plus en plus populaires et utilisés, car, conjointement avec les récents progrès technologiques en informatique, ils nous permettent de reproduire des situations et des environnements d'apprentissage qui dans la vie réelle seraient très coûteux et dangereux à mettre en place. Ainsi, les jeux sérieux se sont imposés comme une méthode innovatrice et efficace pour aider à l'apprentissage de différentes notions chez l'apprenant ; puisqu'ils représentent l'équilibre parfait entre le divertissement propre aux jeux vidéos et l'enseignement de contenu sérieux provenant de diverses disciplines. De plus, leur utilisation dans l'apprentissage et l'acquisition de connaissances a entrainé une augmentation de l'intérêt porté par la communauté scientifique sur cette approche et aujourd'hui le nombre de laboratoires, congrès, conférences, ouvrages scientifiques, etc., dédiés à la recherche sur les jeux sérieux ne cesse de s'accroître.

Du fait de l'augmentation des investissements dans ce secteur (en 2007 le marché des jeux sérieux représente un chiffre d'affaires d'environ $20 million) du à l'intérêt croissant porté sur les jeux sérieux, des techniques et des méthodes avant réservées à la recherche et aux jeux vidéo ont commencé à être utilisées dans la création et le développement de jeux sérieux. Cependant, malgré les avantages apportés par ces méthodes, les concepteur de jeux sérieux affrontent actuellement les mêmes problèmes rencontrés il y plusieurs années dans l'académie et dans l'industrie du jeu vidéo.

Un de ces problèmes, qui est celui qui a été abordé pendant notre stage, est la simulation des comportements de personnages non joueurs lorsque que les actions de l'utilisateur modifient et troublent l'état du jeu ainsi que l'état interne de chaque personnage. Comment créer des personnages qui seront aperçus par l'utilisateur comme réalistes et autonomes ?, quelle méthode ou technique utiliser pour produire rapidement et facilement des comportements réactifs et adaptatifs ?, comment simplifier le travail des concepteurs ?

Pour répondre à ces questions nous vous présentons en détail le travail réalisé au sein de l'équipe Vortex pendant les derniers 4 mois. Nous commencerons en listant les méthodes de simulation comportementale proposées par l'académie ainsi que celles utilisées et développées par l'industrie du jeu vidéo. Ensuite, nous décrirons la théorie derrière la méthode que nous considérons comme capable de résoudre la problématique et les questions énoncées. Puis nous présenterons le concept du jeu sérieux utilisé pour tester l'approche proposée ; et finalement nous décrirons la réalisation de notre jeu et son intégration avec la méthode de simulation comportementale sous le moteur de jeu Unity3D.

Chapitre 1

État de l'art

1.1 Agents virtuels autonomes

Aujourd'hui, grâce aux nombreux progrès technologiques, nous avons à notre disposition des cartes graphiques et des ordinateurs très puissants. Nous sommes, donc, capables de créer et reproduire de larges environnements virtuels de plus en plus complexes, dynamiques et graphiquement très proches du monde réel. Cependant, un monde virtuel visuellement très réaliste ne suffit pas pour offrir aux utilisateurs des expériences uniques et très ressemblantes à ce qui peut se produire dans le monde réel. Nous ne devons pas oublier que même si les mondes virtuels ouvrent à nos yeux une large gamme d'interactions et d'expériences possibles, notre nature humaine attend que certaines lois, organisations, règles et phénomènes propres au monde réel soient présents. Par exemple, nous attendons que ces environnements virtuels soient peuplés par des « créatures virtuelles », appelées aussi agents virtuels, capables d'interagir avec l'environnement qui les entoure, avec eux-même et avec l'utilisateur.

Concevoir, modéliser et animer ces agents virtuels représente un défi majeur pour les concepteurs de mondes virtuels et il est encore plus compliqué lorsque ces agents ressemblent à l'être humain ; car l'utilisateur possède une habilité innée qui lui permet de percevoir tous les détails subtils du comportement et du mouvement humain [MHK99]. Un concepteur pourra facilement décider de scénariser de manière exhaustive -à travers un langage de programmation ou un langage de script-, chacune des actions de l'agent virtuel, les façons dont il agit et interagit avec son monde ainsi que les possibles réponses à chacune des interactions avec l'utilisateur. Cependant, après quelques essais, l'utilisateur sera capable :*a)* d'élucider que le comportement de l'agent est répétitif et peu réaliste, *b)* de prévoir et de prédire chacune de ses actions, et *c)* dans les cas de plusieurs agents, de déterminer que tous se comportent de la même façon et par conséquent qu'ils manquent d'individualité et d'intelligence. Cela donne comme résultat une expérience peu enrichissante et éloignée de ses attentes. De la même façon, scénariser chaque action de chaque agent est un tâche longue et fastidieuse pour le concepteur.

Dû aux difficultés rencontrées pour programmer chacune des actions d'un agent virtuel, d'autres approches ont été proposées ; parmi celles-ci, une des plus utilisées et à laquelle nous allons nous intéresser est la conception et le développement des « agents

autonomes adaptatifs ≫. Cette approche est issue des études provenant de diverses disciplines comme l'animation comportementale, l'intelligence artificielle, la robotique et la vie artificielle [MHK99].

1.1.1 Définition d'agent autonome adaptatif

Le terme agent provient de la philosophie et signifie un acteur, une entité avec des buts et des intentions, qui apporte des changements dans le monde. Le terme agent peut donc être utilisé pour désigner une personne, un phénomène, un animal, un pays ou un robot [Bry01] et ; peut être appliqué dans divers domaines allant de la politique, à l'économie, ou la linguistique jusqu'aux sciences cognitives et l'intelligence artificielle. Pour notre intérêt, nous allons définir un agent comme un système qui habite dans un environnement dynamique et imprédictible et qui essaie d'atteindre un ensemble de buts dans cet environnement. Un but peut être un objectif final, un état particulier, un besoin ou une motivation [Mae94].

Pour pouvoir atteindre ces buts, l'agent doit être :

Autonome : l'agent possède les mécanismes nécessaires pour agir et interagir avec l'environnement sans l'intervention d'autres agents ou du concepteur [Flo03]. L'agent est lui-même capable de rapporter ses perceptions avec ses actions de telle façon qu'il puisse atteindre avec succès ses buts [Mae94]. L'autonomie améliore la viabilité de l'agent dans un environnement complexe, dynamique et imprédictible.

Adaptatif : l'agent est capable d'améliorer sa performance dans le temps, c'est à dire, grâce à ses expériences passées l'agent atteint de mieux en mieux ses buts. Un agent peut être adaptatif de diverses façons : en étant capable de s'adapter facilement à court terme, en étant capable de s'adapter aux petits changements dans l'environnement, en étant capable de répondre aux demandes de l'utilisateur, ou en étant capable de gérer des changements dans l'environnement et des buts plus significatifs à long terme [Mae94].

Situé : l'agent est établi dans l'environnement et il est capable de le percevoir à travers ses sens et d'agir sur lui en utilisant ses actionneurs [Mae94].

Incarné : l'agent a un corps qui lui permet d'interagir avec l'environnement qui l'entoure. C'est important de remarquer qu'un agent ne sera pas complètement incarné s'il n'est pas situé [Flo03].

1.1.2 Architecture de base d'un agent autonome adaptatif

Après l'apparition et la définition d'un agent autonome adaptatif dans le contexte de la simulation comportementale et de l'intelligence artificielle, de nombreux chercheurs comme Thalman, Seels, Reynolds, Blumberg, Terzepoulous et beaucoup d'autres les ont implémentés à leur propre façon. Cependant, un des faits qui est devenu clair est que jusqu'à aujourd'hui il n'existe pas dans la littérature scientifique un ensemble de principes, d'algorithmes et de techniques, appelé aussi architecture, que nous pourrions considérer comme optimal [Mae94] et qui donnerait comme résultat un agent en interaction avec le monde virtuel et capable de montrer un comportement adaptatif, réaliste et efficace.

Néanmoins, comme le but final de la simulation comportementale est de reproduire chez les agents les façons de raisonnement ainsi que les processus mentaux (comme l'attention, la planification, l'apprentissage, etc.) observés chez les humains [Mal97], les sciences cognitives proposent un système cognitif total qui peut être utilisé comme architecture de base lorsque l'on souhaite développer des agents autonomes adaptatif dans les environnements virtuels.

Cette approche cognitive est importante et valide parce qu'il a été montré que ce sont finalement les fonctions cognitives du cerveau qui servent à organiser le comportement d'un individu dans son environnement, ainsi nous avons besoin uniquement de ce système pour expliquer des comportements apparemment très complexes comme la navigation et l'orientation.

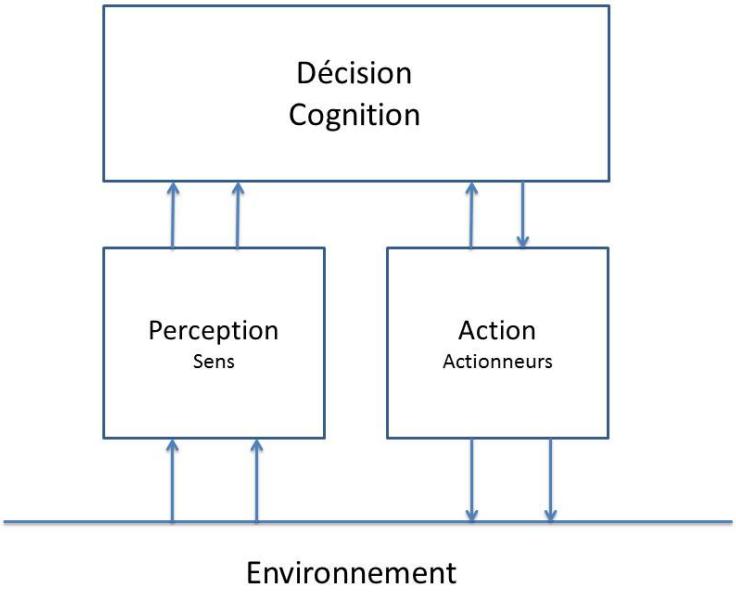

FIGURE 1.1 – Système cognitif total

La Figure 1.1 montre le système cognitif total proposé par Newell. Ce modèle réunit les trois capacités principales dont l'être humain dispose pour structurer sont comportement vis-à-vis de son environnement et des ses intentions et qui sont utilisées comme architecture de base dans la conception et modélisation d'agents autonomes adaptatifs.

Dans ce système les fonctions de chaque composant sont :

- **Perception :** à travers les sens, tant l'humain que l'agent perçoivent l'environnement et, si c'est le cas, traduisent ces perceptions en signaux pour le module de décision.
- **Décision :** en utilisant l'information provenant du module de perception, le système décide quoi faire et quand le faire. Cette décision est prise en prenant

en compte les motivations, les buts, les désirs, etc.
- **Action :** ce module décrit comment exécuter l'action ou plan produit par le module de décision. Ici, les plans sont traduits en actions concrètes comme se déplacer, se nourrir, tourner, etc.

Si nous nous basons sur ce modèle cognitif, toute approche informatique qui a comme but la génération de comportements autonomes et naturels doit mettre en œuvre comme concepts de base de son architecture des agents virtuels autonomes interactifs : une perception de l'environnement, une capacité de prise de décision (traitement) en fonction de perceptions et de buts propres de l'agent et une capacité d'action sur l'environnement [Vey09]. Si nous souhaitons également que les agents soient capables d'enchaîner plusieurs actions afin d'atteindre des comportements décrits d'une façon plus abstraite, nous devons aussi inclure une capacité d'apprentissage ou un module de mémoire.

Grâce a cette architecture de base, nous pouvons réduire le problème de la modélisation des comportements crédibles, cohérents et naturels à une boucle ***Perception-Sélection-Action***, c'est-à-dire, l'agent perçoit l'état du monde virtuel (phase de perception) ; partant de cet état et de son propre état interne (buts, intentions, etc.) il décide l'action ou ensemble d'actions à exécuter (phase de sélection) et finalement il agit (phase d'action). Ces actions vont modifier l'état de l'environnement, l'état interne de l'agent et ainsi permettre d'agir sur le comportement de l'utilisateur [Ver06].

1.1.3 Avantages des agents autonomes adaptatif

L'utilisation d'agents autonomes adaptatifs permet la simplification du travail des concepteurs, car la conception d'un monde virtuel se résumera à la création d'un environnement (une maison, une ville, une forêt, une usine, etc.) et au placement d'agents possédant des règles de comportement parmi lesquelles sélectionner chacune de ses actions. Cela produira des animations et simulation plus crédibles et riches pour l'utilisateur avec moins d'interventions des concepteurs [MHK99], [MTT04].

Également, les mondes virtuels peuplés par ce type d'agents vont profiter de l'émergence de comportements plus complexes grâces aux interactions autonomes entre l'agent et l'environnement.

1.2 Modélisation des comportements autonomes

La modélisation des comportements autonomes chez les humains et aussi chez les animaux est aujourd'hui une tâche complexe, qui prend beaucoup de temps et qui est abordée par différentes disciplines comme la psychologie, l'intelligence artificielle, l'infographie... avec un champ d'application allant de l'éthologie aux sciences sociales [dS06]. Tous les modèles, approches et architectures qui vous seront décrites par la suite cherchent à développer comme l'a exprimé N. Badler : des agents virtuels autonomes (humains ou non) qui ont des mouvements, des réactions, une prise de décision et des interactions qui semblent naturels, appropriés et dépendants du contexte.

Pour la clarté de la lecture, nous vous rappelons que le terme agent et le terme entité seront utilisé indistinctement lorsque nous parlerons des agents autonomes adaptatifs.

1.2.1 Architectures cognitives

Lorsque l'intérêt se centre sur la modélisation de comportements autonomes d'entités qui ressemblent à l'être humain, l'approche proposée par les sciences cognitives cherche à définir des modèles et des architectures qui essayent d'unifier les fonctions les plus représentatives de l'esprit humain : la perception, le langage, la mémoire, etc.

En général on peut distinguer deux types d'architectures : *(1)* Celles fondées sur l'idée que pour concevoir une entité autonome, appelée aussi agent autonome, il n'est pas nécessaire d'avoir une représentation des connaissances de l'agent parce qu'il suffit de le doter d'un ensemble de compétences élémentaires, ainsi que d'un mécanisme de contrôle et de la capacité d'exploiter les informations et propriétés provenant de l'environnement dont il fait partie ; *(2)* Celles qui cherchent à modéliser des entités (agents) capables d'élaborer des plans complets afin d'atteindre un but. Cela requiert des entités dotées d'une représentation des connaissances et d'un contrôle de l'action.

Parmi toutes les architectures existantes, je vais vous présenter les plus emblématiques en informatique :

SOAR - State, Operatator and Résult

Proposée par J. Laird et al. en 1986 et montrée dans la Figure 1.2, SOAR cherche à modéliser les processus cognitifs qui ont lieu au moment d'effectuer différentes tâches. Afin d'effectuer une tâche, le système implémentant SOAR doit se créer une représentation de cette tâche qui le guidera dans son accomplissement. Cette représentation est contenue dans une mémoire de travail sous la forme d'une hiérarchie d'états à atteindre et d'opérateurs permettant de passer de l'état courant vers un nouvel état.

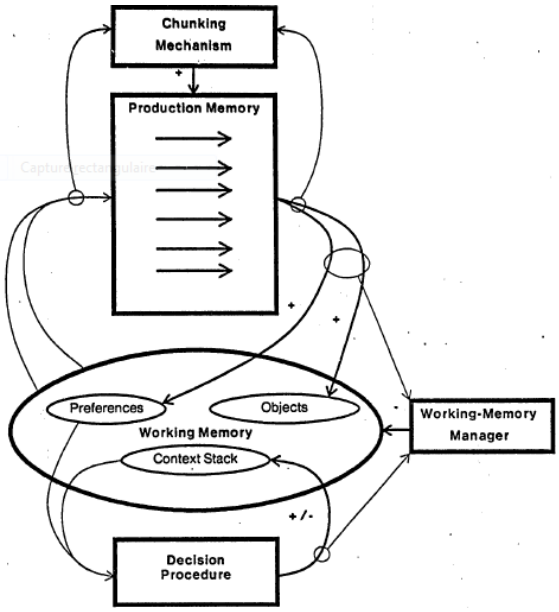

FIGURE 1.2 – Structure de l'architecture SOAR

SOAR a comme hypothèse générale que tous les comportements délibérés orientés buts peuvent être modélisés (représentés) par la sélection et l'application d'opérateurs à des états [Don04]. La sélection d'opérateurs à appliquer se fait en utilisant des règles de productions (pair condition-action) contenues dans une mémoire principale ainsi que des connaissances portant sur les préférences d'une action dans un état donné parmi toutes les actions possibles.

L'apprentissage se fait par l'expérience, c'est-à-dire qu'une fois qu'une tâche a été accomplie des nouvelles règles de production seront crées et stockées dans la mémoire de travail. L'application la plus représentative de cette architecture à été faite dans un des premiers agents tuteurs virtuels développés, i.e STEVE - Soar Training Expert for Virtual Environments.

ACT-R - Adaptative Control of Thought-Rational

Proposée par J.R. Anderson en 1993 et montrée dans la Figure 1.3, ACT-R a comme objectif de proposer une théorie générale de la cognition en mettant l'accent sur l'apprentissage procédural (développement d'habilitées). [BLCR09].

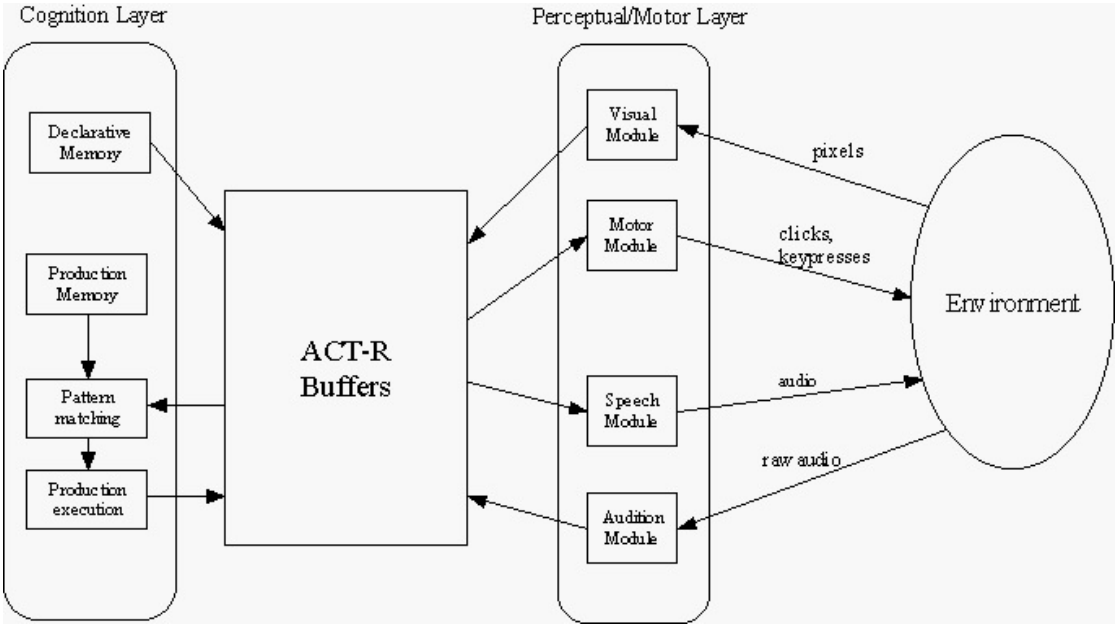

FIGURE 1.3 – Structure de l'architecture ACT-R

Contrairement à SOAR, ACT-R est une architecture modulaire où chaque module traite un type spécifique d'information et à chaque cycle de décision une et une seule règle de production peut être appliquée. ACT-R est aussi une architecture centralisée, c'est-à-dire que toute la coordination de l'accomplissement d'une tâche et par conséquent de la réalisation d'un comportement est faite par un système central qui ne reçoit qu'une part de toute l'information traitée par chaque module. ACT-R peut être utilisé pour développer des simulations d'un grand nombre de phénomènes cognitifs [Don04].

1.2.2 Système multi-agents

Issues de l'intelligence artificielle les systèmes multi-agents sont largement utilisés dans les domaines où l'intérêt principal est de modéliser l'émergence des comportements complexes à partir des interactions entre plusieurs individus, chacun munit de comportements simples (se déplacer, se nourrir, etc.).

[DPP09] : Nous appelons agent une entité réelle ou abstraite qui est capable d'agir sur elle-même et sur son environnement, qui a une représentation partielle de cet environnement, qui peut communiquer avec les autres agents d'un univers multi-agents et dont le comportement est la conséquence de ses observations, ses connaissances et ses interactions avec les autres agents.

Il existe deux types d'agents : *(1)* Agents réactifs qui se caractérisent par des comportements stimuli-action, c'est-à-dire, ses actions sont choisies et exécutées selon les événements perçus dans l'environnement ; les agents de ce type peuvent ne pas être intelligents de manière individuelle, mais produisent des comportements intelligents en collectivité. *(2)* Agents cognitifs qui sont capables de coopérer avec l'autre afin de résoudre un problème. Pour cela ils ont et maintiennent une représentation commune de l'environnement dont ils font partie. Contrairement aux agents réactifs, les agents cognitifs sont plus autonomes et possèdent un niveau d'expertise et de connaissance plus élevé.

Les architectures multi-agents existantes sont :

Modèle BDI - Beliefs, Desires, Intentions

Ce modèle s'inspire du travail de M.E. Bratman sur le raisonnement pratique de l'être humain. Les concepts de base sont :

- Beliefs/Croyances : La connaissance que l'agent croit avoir sur l'état du monde qui l'entoure.
- Desires/Désirs : Les motivations et les préférences de l'agent.
- Intentions : L'ensemble des plans que l'agent peut exécuter pour satisfaire ses désirs.

Un agent rationnel BDI, en supposant qu'il possède un certain nombre de plans, opère ainsi [Don04] :

```
Boucle infinie
 Observer le monde
 Mettre à jour le modèle interne du monde
 Délibérer sur quel doit être le prochain désir à réaliser
 Raisonner afin de trouver un plan pour satisfaire le désir
Fin boucle
```

Une fois que le plan a été sélectionné il devient une intention et toutes les actions qui l'intègrent sont exécutées pas à pas. Un plan peut consister à : ajouter un nouveau désir, modifier les croyances de l'agent ou exécuter une action en particulier.

PECS - Physical conditions, Emotional state, Cognitive capabilities, Social status

Proposé par B. Schmidt en 2000 PECS a comme intention la modélisation du comportement dans un contexte social.

PECS est composé d'une architecture en trois couches qui lui permet de prendre compte des états physique, émotionnel, cognitif et social de l'agent afin de modéliser tous les niveaux de comportement existants.

- Couche de perception qui est responsable du traitement de toute l'information provenant de l'environnement
- Couche interne qui modélise chacun des états internes de l'agent
- Couche externe qui comprend les actions et comportements de l'agent.

Brahms

Basé sur la théorie de l'activité de J. Clancey, Brahms est un langage orienté agent utilisé pour modéliser et simuler les activités humaines. Il est structuré autour des concepts suivants : le monde est composé d'*agents* qui peuvent appartenir à plusieurs *groupes*, ces derniers bénéficiant des comportements des agents qui les composent, et des *objets* (représentations d'artefacts) dépourvus d'un comportement propre et qui réagissent uniquement aux changements d'état du monde.

Chaque agent a un ensemble de *croyances* susceptibles de changer à cause des événements qui surviennent dans le monde. À l'opposé les *faits* sont des états du monde connus par tous les agents. Les activités quant à elles ont une durée fixée ainsi que des états conditionnels ; cela limite la réalisation d'une activité aux moments où les croyances de l'agent équivalent à l'état conditionnel de l'activité.

L'association entre une tâche et son état conditionnel est faite par le *cadre de travail*. Au contraire *le cadre de pensée* permet la modélisation du déroulement d'un raisonnement exécuté par un agent ainsi que la description des conséquences liées aux états conditionnels. Grâce au cadre de pensée il est possible de déduire des nouvelles croyances pour un agent. Pour finir, il existe aussi une *géographie* qui indique les endroits dans lesquelles un agent peut exécuter une activité ainsi que se déplacer.

1.2.3 Animation comportementale

L'animation comportementale fait allusion aux différentes méthodes dont l'informatique se sert afin de modéliser les comportements des entités qui peuplent un monde virtuel et qui ont comme caractéristiques communes la capacité de percevoir ce monde ainsi qu'un certain niveau d'autonomie. En général, toutes les approches proposées par l'animation comportementale ont comme finalité de sélectionner la meilleure action à réaliser dans une certaine situation parmi l'ensemble de toutes les actions possibles. Si l'on est capable de bien réaliser cette sélection, on peut modéliser toute sorte d'individus vivants (plantes, animaux et êtres humains) [Don04].

Pour accomplir cet objectif l'animation comportementale prend comme base la boucle **Perception-Décision-Action** mentionnée auparavant : une entité perçoit l'environnement, décide de la prochaine action à exécuter et agit sur le monde. C'est

important de remarquer que le processus décisionnel réalisé par l'entité peut être réactif ou cognitif.

Avec un processus décisionnel réactif, l'entité choisit une action de manière inconsciente, c'est-à-dire, sans l'intervention d'aucune connaissance ni expérience. À l'opposé, dans un processus cognitif toutes les connaissances et expériences antérieures de l'entité influent sur sa décision ; ce type de processus permet de sélectionner une suite d'actions cohérentes visant à atteindre un but donné et l'entité est alors capable de raisonner sur les conséquences de ses actions [Don04].

Systèmes réactifs

Ils permettent de modéliser des comportements modérément complexes, sans faire appel à une modélisation des connaissances. Ces systèmes sont caractérisés par l'absence d'une projection dans le temps au moment de décider l'action suivante, c'est-a-dire, que l'entité ne regarde jamais les conséquences de ses actions. Les systèmes réactifs les plus utilisés sont :

- **Les systèmes stimuli-réponse** : Basés sur les idées issues de l'éthologie, ils considèrent qu'un comportement est une réponse aux interactions directes avec l'environnement. Ce type de systèmes ont deux composants de base : les capteurs, utilisés pour percevoir l'environnement, et les effecteurs, utilisés comme moyen pour agir sur le monde ; ces deux éléments sont reliés par un réseau de nœuds intermédiaires qui transforment l'information provenant des capteurs en commandes qui seront exécutés par les effecteurs.
  formels
  Le connexionnisme en s'inspirant de l'organisation du cerveau propose la représentation des comportements sous la forme d'un réseau de neurones formels ; chaque neurone correspond à une fonction d'activation de la forme $y = f(x)$ où x représente les capteurs et y les effecteurs.
  Ce qui est intéressant dans ces systèmes c'est la possibilité d'utiliser des algorithmes d'apprentissage qui permettent de configurer, à partir d'un ensemble d'exemples, le réseau automatiquement en fonction d'un comportement espéré.

- **Les systèmes à base de règles** : Ces systèmes utilisent aussi des capteurs afin de percevoir les informations provenant de l'environnement et/ou l'état interne de l'entité virtuelle ; ces perceptions sont prises en compte pour sélectionner la prochaine action à effectuer. Dans ces systèmes un comportement correspond à une règle sous la forme $conditions => action$; ainsi l'action à effectuer sera choisie parmi l'ensemble des actions dont les conditions sont satisfaites par les perceptions provenant des capteurs. La principale différence entre deux ou plusieurs systèmes à base de règles est le mode utilisé pour représenter ces règles.
  En général, les représentations les plus utilisées sont :
  – Un ensemble de règles simples de type $si...alors....$
  – Un arbre de décision où les feuilles correspondent aux actions et les nœuds sont des experts en charge d'effectuer un choix entre tous leur sous-nœuds [Lamarche

et al. Traité 5].

- **Les automates à états finis** : Cette approche modélise les comportements comme des enchaînements conditionnels d'actions. Pour cela elle se sert de deux composants : *(1) des états* qui correspondent à une tâche ou action unitaire ; *(2) des transitions* qui correspondent aux conditions d'enchaînement entre deux états, ces conditions équivalent au contexte et au dernier état attendu. Comme l'a exprimé [LD09] il existe plusieurs typologies de systèmes permettant de décrire et gérer l'exécution de comportements via des automates :
  - *Les piles d'automates* : lorsqu'un automate demande l'exécution d'un autre, ce premier est empilé et le dernier est exécuté. Une fois que le dernier arrive à son état terminal, le premier est dépilé et reprend son exécution.
  - *Automates parallèles* : plusieurs automates peuvent être exécutés simultanément donnant comme résultat des comportements complexes.
  - *Automates parallèles hiérarchiques* : un comportement correspond à une hiérarchie d'automates qui fonctionnent de manière simultanée. Ici l'automate père peut soit superviser le fonctionnement de ses automates fils soit filtrer et/ou fusionner les résultats provenant de l'exécution de ceux-ci.

Systèmes cognitifs et orientés buts

Ils nécessitent une représentation des connaissances de l'entité afin de calculer un enchaînement d'actions permettant d'atteindre un but donné. Les différents modèles qui vous seront présentés diffèrent par leur coût de calcul, leur façon dont les connaissances sont représentées, leur réactivé aux changements dans l'environnement ainsi que leur capacité à exploiter des opportunités [LD09].

- **Le calcul situationnel** : Proposé par McCarty et P.J Hayes en 1969, il permet de décrire des mondes complexes ainsi que les comportements (actions) des entités qui les peuplent. Pour cela, il se base sur les éléments suivants :

  Situations : états complets du monde virtuel à un instant donné.

  Les fluents : fonctions qui permettent de décrire une propriété dynamique du monde.

  Les causalités : relations cause-effets utilisées pour déduire l'état d'un fluent (propriété) en fonction de l'état d'autres fluents.

  Les actions : sous la forme $preconditions => effets$ elles modifient une situation.

  Les stratégies : enchaînements d'actions en fonction de certains raisonnements.

  Les connaissances : elles modélisent si une entité connaît ou non l'état de certains fluents du monde.

  À partir de ces éléments, une phase de planification à lieu donnant comme résultat la suite d'actions permettant de générer une situation du monde qui satisfait un but donné. Pendant cette phase tous les mondes possibles sont explorés, cela suppose un temps de recherche considérable dans les cas où l'ensemble des actions

possibles est très large.

- **STRIPS - STandford Research Institute Planning System** : Dû à la contrainte de temps de recherche présente dans le calcul situationnel, R.E. Fikes et al. ont proposé en 1971 une version simplifiée de ce calcul. Dans cette nouvelle version la notion des *fluents* disparaît et l'état du monde est exprimé en fonction des *faits* présents ou absents dans une situation particulière. Cette modification rend les algorithmes utilisés dans la recherche du monde attendu plus efficaces.

- **Les réseaux de tâches hiérarchiques - HTN** : Dans cette approche la planification du comportement d'une entité est faite hiérarchiquement en prenant en compte trois concepts de base :
  - *Tâches* : elles équivalent aux buts à atteindre. Une tâche correspond à la satisfaction d'un ensemble de propriétés dans le monde, à la réalisation d'un ensemble d'actions ou à une combinaison des deux [LD09].
  - *Méthode* : c'est la manière avec laquelle il est possible d'accomplir une tâche. Une méthode peut être une suite de sous-tâches ou d'actions. Comme dans les approches déjà décrites, avant d'exécuter une méthode, un certain état du monde (conditions) doit être vérifié.
  - *Actions* : sous la forme $preconditions => effets$ elles seront immédiatement réalisées si leur pré-conditions sont vérifiées.

  La planification consiste à décomposer la tâche cible en sous-tâches en appliquant les méthodes proposées. Cette décomposition donne comme résultat un type particulier d'arbre nommé « arbre et/ou », où chaque nœud peut correspondre soit à une succession de « tâches et/ou » (nœud et) soit au choix entre les méthodes qui peuvent être utilisées pour la réalisation d'une même tâche (nœud ou). Le comportement final de l'entité consiste en une suite de « nœuds et/ou» de telle façon que les actions décrites dans chaque nœud permettent d'accomplir la tâche cible.

- **Les mécanismes de sélection d'actions** : Le fonctionnement de ces systèmes se base sur une rapide réaction face aux changements de l'environnement ainsi que sur la possibilité d'utiliser au mieux ces changements. Le comportement final de l'entité est construit de façon incrémentale, c'est-à-dire, à chaque instant l'action qui promet la meilleure convergence vers le but final est choisie en prenant en compte les changements de l'environnement et l'acquisition de nouvelles connaissances[LD09].

1.2.4 Bilan

Dire qu'une méthode est meilleure qu'une autre n'est pas possible, car chacune possède des caractéristiques différentes et modélise différents concepts utiles à la description et la gestion du comportement d'une entité virtuelle autonome [LD09] ; néanmoins il est possible de lister quels sont leurs point positifs et leurs limites.

Les architecture cognitives SOAR et ACT-R ont été construites selon la même idée ; cependant il existe des différences considérables entre les deux : en premier lieu SOAR

est limitée par le fait que seulement un opérateur d'action peut être sélectionné à la fois, de son côte ACT-R ne permet que la sélection d'un seul but à tout moment, ces deux contraintes font de l'accomplissement d'un but ou d'une tâche complexe un processus demandant un temps de calcul considérable ; en deuxième lieu, même si les règles de productions de ces deux architectures permettent l'exécution simultanée de plusieurs actions, les combinaisons d'action sont mieux gérées par SOAR que par ACT-R ; enfin ACT-R base son apprentissage sur une analyse rationnelle de l'information tandis que l'apprentissage avec SOAR évolue au fur et à mesure que les buts sont résolus [BLCR09].

Le principal avantage des systèmes multi-agents est la grande diversité de comportements complexes qui peuvent émerger à partir des interactions entre entités simples (dans le sens où les actions qu'elles exécutent sont basiques et primitives) et l'environnement qui les entoure. Également, ce sont des systèmes hautement réactifs aux changements de l'environnement, adaptatifs car le choix d'actions est incrémentielle ce qui permet l'utilisation et la génération de nouvelles connaissances à tout moment et opportunistes parce que chaque choix fait cherche à utiliser le mieux possible les changements provenant de l'environnement. Cependant, comme l'exprime [DPP09] les systèmes multi-agents souffrent de ce que l'on pourrait appeler une désincarnation des acteurs ainsi que d'une vision très abstraite du monde les environnant.

Les systèmes réactifs sont avantageux parce qu'ils n'ont pas forcément besoin d'une représentation abstraite de l'environnement et des connaissances de l'entité, cela permet d'obtenir de très bons résultat lors de la simulation de comportements simples (ils n'ont pas besoin d'un enchaînement de multiples actions) avec un coût de calcul faible. De plus, si l'environnement simulé est hautement dynamique l'utilisation de systèmes réactifs est privilégiée [LD09].

Les principales différences entre les typologies de systèmes réactifs présentées sont : au contraire des systèmes à base de règles et des automates, les systèmes stimuli-réponse n'ont pas besoin de connaître le fonctionnement du processus qu'ils simulent, cela leur permet de générer automatiquement des entités capables de réaliser des tâches complexes à partir d'exemples mais les rend aussi difficiles à interpréter et à entretenir. Au contraire, les systèmes à base de règles, en particulier les automates, sont facilement interprétables et par conséquent plus utilisables. Les systèmes à base de règles permettent aussi des descriptions plus abstraites et ils peuvent très bien simuler des comportements où la cohérence temporelle n'est pas importante. À l'opposé, les automates permettent de décrire des comportements où une certaine cohérence temporelle dans l'enchaînement des actions est importante [LD09].

Pour finir, les systèmes cognitifs et orientés buts permettent de générer des plans complets d'action en utilisant une représentation abstraite de l'environnement. Pour faire cela ces systèmes supportent des processus décisionnel beaucoup plus élaborés et complexes.

Les principales différences entre les typologies de systèmes cognitifs présentées sont : le calcul situationnel, STRIPS et les HTNs qui supposent que l'unique source de modification de l'environnement sont les entités, cela les rendant peu réactifs aux changements dans l'état de l'environnement ainsi que moins efficaces dans les cas où l'environnement est peuplé par plusieurs entités.

Les mécanismes de sélection d'action grâce à un processus décisionnel incrémentale

n'ont pas ce type de contraintes. Le calcul situationnel et STRIPS ont 100% de capacité à satisfaire un but, ce qui n'est pas le cas des autres typologies [LD09]. Finalement, chaque système utilise un modèle d'expression de but différent, alors que le calcul situationnel et STRIPS utilisent un modèle de formules logiques, HTN part d'une notion de tâche pour décrire les buts du système cela le rendant moins flexible par rapport aux autres typologies.

1.3 La simulation comportementale dans les jeux vidéo

Comme cela a déjà été énoncé, les méthodes de simulation comportementale décrites dans la section précédente peuvent être appliquées dans tous les domaines qui font face à la problématique de génération d'agents autonomes capables de montrer des comportements pertinents dans un contexte donné, réalistes et intelligents. Un de ces domaines est l'industrie des jeux vidéo où, depuis l'apparition de processeur graphiques de plus en plus puissants et où, du fait de la nature compétitive propre au secteur, l'inclusion des ennemis, des collègues ou en général des personnages capables de montrer des actions et comportements plus intelligents et en rapport avec les actions du joueur est devenu un facteur clé dans le succès ou l'échec des nouveaux jeux [JW01], [LvL01].

Au-delà de cette importance économique, il est intéressant de remarquer que les jeux vidéo ont évolué pour devenir un moyen de divertissement majeur, offrant aux joueurs un moyen alternatif pour ressentir un sentiment de plaisir, un sentiment de **Flow** [Che07]. Le terme **Flow** a été introduit par Mihaly Csikszentmihalyi dans les années 70 pour essayer d'expliquer le sentiment de joie chez l'homme ainsi que pour représenter le sentiment de complète implication dans une activité avec un grand niveau de divertissement et d'accomplissement. Lorsque le joueur se trouve dans une expérience de **Flow** il perd la notion de temps et de soucis ; de la même façon, le niveau d'attention qu'il concentre sur le jeu lui permet de maximiser sa performance et le sentiment de contentement qu'il expérimente [Che07]. Il a été démontré que dans certains types de jeux, surtout ceux peuplés par des personnages différents du joueur (des personnages non joueurs), les comportements des agents influencent considérablement l'expérience du joueur et par conséquent son expérience de **Flow**.

Dans le jeux vidéo on ne cherche pas à développer des agents invincibles face auxquels il est impossible de gagner ou des agents tellement prévisible qu'ils ne représentent pas un challenge pour le joueur [And03] ; le but final est toujours de développer ou d'utiliser une méthode de simulation de comportements capable de produire des agents crédibles et aussi amusants que possible [Nar04]. Ce qu'on souhaite, c'est un équilibre parfait entre le challenge offert par le jeu et la capacité du joueur.

Dans le processus du développement de personnages non joueurs, les problèmes les plus rencontrés sont : la recherche et la planification du chemin, la sélection d'actions (par conséquent la simulation comportementale) et le pilotage et contrôle du mouvement [And03]. Nous allons nous intéresser encore une fois à la simulation comportementale, pour cela, nous allons donner des exemples d'application des méthodes déjà décrites en listant les pour et le contres rencontrés par l'industrie des jeux vidéo ainsi qu'en décrivant en quelques mots des méthodes utilisées dans certains jeux et qui n'ont pas été traitées

dans la section précédente. De la même façon, afin de donner au lecteur une perspective global des trois problèmes énoncés, nous allons donner une brève explication de la recherche et de la planification du chemin et, du pilotage et du contrôle du mouvement.

1.3.1 La recherche et la planification du chemin

Les personnages doivent interagir dans des environnements complexes, dynamiques et possédant une géographie étendue, par exemple ils doivent parcourir différents types de dispositions spatiales (chambres, labyrinthes, plateformes, etc) d'une façon intelligente et réaliste ; c'est à dire, en prenant des raccourcis si ils existent ou en évitant des obstacles [Nar04]. Un personnage capable de traverser des objets ou qui arrête son déplacement car il a trouvé un obstacle, qui est franchissable depuis la perspective de l'utilisateur, va créer une brèche dans l'illusion de comportement intelligent et adaptatif que nous essayons de produire.

Il existe diverses solutions à ce problème, certaines sont proposées par l'intelligence artificielle et les autres correspondent aux méthodes empiriques et approximatives utilisées par l'industrie des jeux vidéo. Ces dernières se sont montrées plus efficace en temps de calculs dans des environnements moins complexes et avec moins d'obstacles [Cav00]. Dans le cas des environnement très larges et peuplés par des obstacles, des personnages non joueurs et des joueurs, la méthode la plus utilisé est l'algorithme A* illustré dans la Figure 1.4.

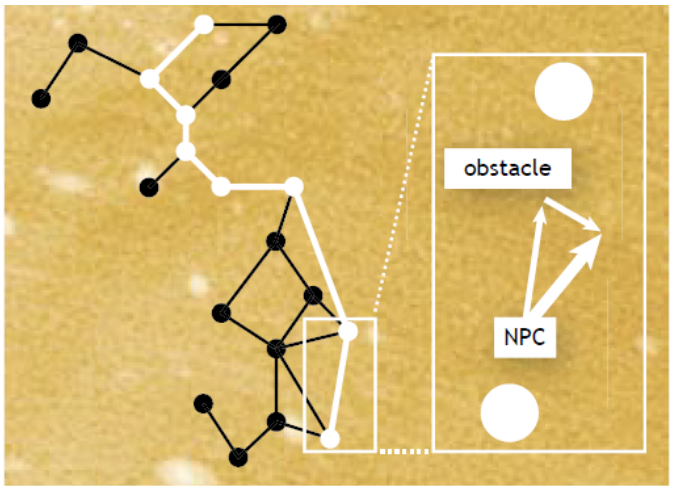

FIGURE 1.4 – Exemple algorithme A* et pilotage du mouvement

Cet algorithme cherche le chemin optimal, dans le cas de jeux vidéo c'est le chemin le plus court, entre un nœud initial et un nœud destin prédéfini. Parmi les titres de jeux les plus connus, des jeux comme Half Life, Descent 3, Quake III l'utilisent pour le déplacement des personnages non joueurs. [LvL01]

1.3.2 Le pilotage et contrôle du mouvement

Pour suivre le chemin trouvé par l'algorithme A*, ainsi que pour donner aux personnages la capacité de parcourir leur environnement d'une manière très proche à celle perçut par l'esprit humain, nous avons besoin de ce qu'on appelle le pilotage et contrôle du mouvement. La technique la plus utilisé a été proposée par C. Reynold [Rey99] et consiste à calculer un vecteur avec une direction ainsi qu'une vitesse et une force pour chaque type de mouvement que nous souhaitons inclure dans notre personnage (éviter des obstacles, suivre un leader, suivre un groupe, échapper d'un ennemi, etc.). Les calculs faits pour chaque comportement seront finalement additionnés pour produire un seul vecteur final qui sera appliqué au mouvement et pilotage du personnage [Nar04] comme il est montré dans la Figure 1.4.

1.3.3 Application et nouvelles méthodes de simulation comportementale

Techniques à base de règles

Ce sont les techniques les plus utilisées par le jeux vidéo car elles sont très proches de la façon dont l'être humain raisonne lorsqu'il doit résoudre un problème ; elles permettent de décomposer des comportements très complexes dans des comportement plus petits et simples à aborder [BKKP06]. De plus, elles sont facilement implémentées et les comportements obtenus sont robustes et fiables [And03].

1. **Les automates à états finis :** appelés aussi machines à états finis, ils sont la méthode de simulation comportementale la plus utilisé dans le jeux vidéo pour plusieurs raisons : leur simplicité, ils peuvent être mis en œuvre efficacement, leur taille est en général modérée et les entretenir ne pose pas de grands problèmes. Nous pouvons les trouver implémentés dans jeux comme Quake et Team Buddies [Cav00]. Malgré ces avantages, le principal problème des automates à états finis est qu'au fur et à mesure que nous ajoutons de nouveaux états et comportements, ils peuvent devenir très complexes et difficiles à gérer. Au contraire, lorsque nous avons des automates très simples, le joueur peut facilement prédire toutes les actions des personnages [And03]. La Figure 1.5 montre l'automate qui dirige le comportement d'un personnage non joueur qui essaiera de prendre le trésor lorsque un autre personnage de type monstre est mort ou absente.

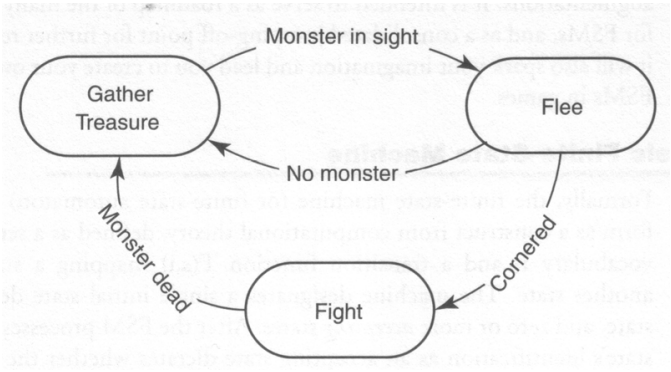

FIGURE 1.5 – Exemple d'une machine à états finis

2. **Les machines d'état floue :** sont une permutation des machines à états finis. A l'inverse de ces dernières où un état est présent (vrai) ou absent(faux) à un moment donné, chez les machines d'état floue un état peut avoir une valeur intermédiaire (pas complètement vrai, pas complètement faux). Cette particularité fait qu'au moment où l'agent choisira son prochain comportement il le fera parmi plusieurs comportements actifs et par conséquent il sera moins prévisible. Un autre avantage de cette méthode est qu'elle permet de réduire la complexité propre aux machines à états finis très volumineuse, car une grande variété des comportements et d'actions peuvent être programmées avec moins d'états. Un des inconvénient de cette approche est que son implémentation est légèrement plus compliquée et moins évidente pour les concepteurs. Des jeux très connus comme *The Sims* et *Civilisation : Call to Power* l'utilisent pour la simulation comportementale de leurs personnages non joueurs [And03].
3. **Les arbres des décisions :** comme les machines d'état finis, les arbres de décisions sont rapides, faciles à implémenter et à comprendre, ainsi que largement utilisés pour la simulation comportementale, cependant il peuvent être facilement prévisibles pour le joueur. Ils sont formés par des points de décision connectés, incluant la racine, qui contient un ensemble d'actions possibles. Chaque fois que l'agent doit choisir une action, il commence par parcourir l'arbre en décidant entre un nœud et un autre selon la connaissance actuelle qu'il a du monde virtuel. Ce processus continue, jusqu'à ce que l'agent n'ait plus de décisions à considérer et l'action trouvée dans une des feuilles de l'arbre est alors exécutée [Nar04], [Mil06]. La Figure 1.6 montre un arbre de décision où le personnage décidera par exemple d'attaquer si l'ennemi est visible et à moins de 10 mètres. La Figure 1.7 montre le même arbre avec une action déjà choisie.

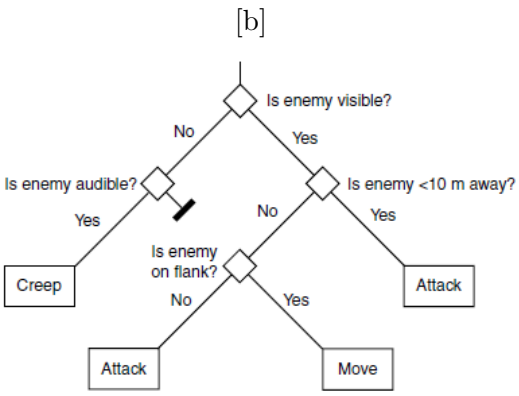

FIGURE 1.6 – Exemple arbre de décision

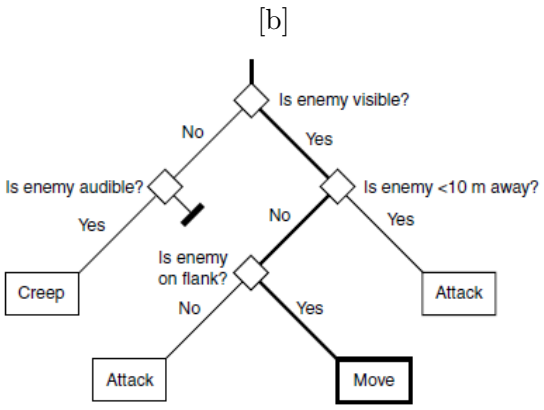

FIGURE 1.7 – Exemple arbre de décision avec une action choisie

Techniques issues du Machine Learning & Machine Intelligence

Dans les dernières années l'intérêt pour les personnages non joueurs capables d'apprendre de leurs erreurs ainsi que de reconnaitre le style du joueur afin de s'y s'adapter a augmenté considérablement ; l'industrie cherche à développer des personnages capables d'apprendre à jouer chaque fois mieux [Cav00]. Les techniques les plus utilisées sont :

1. **Les réseaux de neurones :** s'inspirent de la biologie et essaient de simuler le fonctionnement du cerveau. Ils sont représentés par des nœuds simples connectés et qui exécutent chacun le même processus. Les résultats provenant de chaque nœud permettent au réseau d'apprendre et de s'adapter pendant l'exécution du jeu [And03]. Généralement, ils sont utilisés dans les jeux de stratégie et d'action comme *Heavy Gear*.
2. **Les arbres des décisions extensibles :** sont des arbres de décisions capables de grandir à fur et mesure qu'ils apprennent des nouvelles informations. L'avantage de ce type d'arbre est qu'après être généré il sera capable de garder des informations concernant des situations déjà affrontées et les résultats de ses actions, cela lui permettant de retenir la meilleur action à exécuter dans une situation future similaire à une situation déjà rencontrée. Ils sont les plus utilisés dans les jeux où une méthode d'apprentissage est nécessaire et nous pouvons les trouver dans des jeux comme *Black and White* [And03].
3. **Les techniques évolutionnistes :** sont les moins utilisées car elles prennent très longtemps avant d'aboutir à un niveau d'apprentissage et de performance acceptable. Elles consistent en un ensemble de solutions capables d'évoluer dans le temps à travers l'utilisation des méthodes de sélection et d'évolution inspirées de la génétique.

Les techniques qui viennent d'êtres présentées ne sont pas très utilisées car elles demandent beaucoup de ressources pour être implémentées, perfectionnées et testées. Ainsi, comprendre les comportements résultants et pourquoi ils ont été générés représente

une tâche en général très complexe pour les concepteurs car ces comportements sont difficiles à tester et modifier. Enfin, en utilisant ces techniques le risque de produire des agents avec des comportements peu intelligents et réalistes est plus élevé [Nar04].

Chapitre 2

Contributions du stage

2.1 Objectifs et problématique du stage

2.1.1 Contexte

Jusqu'à présent, nous avons fait une révision rapide des problèmes liés à la simulation des comportements des agents peuplant un environnement virtuel lorsque ces agents sont en interaction directe avec des utilisateurs, ainsi que les possibles solutions provenant de l'académie et de l'industrie. Nous avons vu que la difficulté principale se présente lorsque l'agent doit choisir parmi ses actions disponibles celle qui est le plus en accord avec l'état du monde virtuel, avec l'interaction actuelle entre l'agent et l'utilisateur et avec son état et ses intentions. Ce difficulté augmente lorsque nous souhaitons que les comportements montrés par les agents et que tous les types d'interactions présentés (agent-environnement, agent-agent, agent-utilisateur) soient en accord avec le scénario défini par le concepteur du monde virtuel en gardant toujours sa nature réactive et adaptative.

C'est dans ce contexte que les travaux que j'ai réalisé tout à long de mon stage au sein de l'équipe Vortex à l'IRIT s'inscrivent. Concrètement, le cadre de mes études et mes travaux est la génération de la scénarisation, de la simulation et de l'interaction des agents autonomes dans les environnements virtuels interactifs scénarisés adaptatifs. Lorsque nous parlons de la génération des environnements ou des agents scénarisés adaptatifs interactifs, nous faisons référence à la conception d'un monde virtuel et de personnages selon un scénario créé par le concepteur du monde virtuel ou par un expert dans un certain domaine. Dans ce type d'environnement, les interventions de l'utilisateur vont influencer la façon dont le scénario se déroule en respectant toujours le scénario défini par le concepteur et, tant l'environnement que les personnages seront capables de s'adapter d'une telle façon que leur exécution répondra aux demandes d'interactions de l'utilisateur, prendra en compte les changements introduit par celui et suivra le scénario défini.

2.1.2 Objectifs

En prenant en compte le contexte décrit dans les paragraphes précédents, notre objectif principal est de proposer une méthode de simulation comportementale capable de produire des comportements qui, depuis la perspective de l'utilisateur, soient perçus comme crédibles, intelligents et réactifs ; et qui depuis celle du concepteur soient faciles à définir et à tester et en accord avec le scénario défini. D'autre part, pour donner une continuité aux intérêts et projets actuels de l'équipe Vortex, la méthode que nous allons proposer pour la génération des agents autonomes doit aussi rester :

- facile à utiliser pour les personnes qui possèdent peu ou aucune base en programmation ;
- assez générique pour qu'elle puisse être utilisée dans d'autres contextes et projets ;
- et facilement extensible pour que des actions et comportements plus larges puissent être intégrés.

Un dernier objectif, au début secondaire mais qui s'est par la suite révélé determinant pour les contributions que nous proposons, est de regarder les concepts du Game Design et les outils permettant de créer des jeux et des simulation comme : « game factory », éditeur de scénario, description de l'interaction, etc.

2.1.3 Contributions

Dans la suite de ce document nous proposerons une méthode appelée **Arbres des comportements** (Behavior trees en anglais) issue de l'industrie des jeux vidéo [Isl05] et que nous considérons très utile pour l'implémentation des agents autonomes adaptatif scénarisés. Nous proposerons un concept de jeu sérieux afin de tester les limites et l'adéquation des arbres de comportement ; ensuite nous décrirons le processus de développement du jeu ainsi qu'une implémentation des arbres de comportements prenant en compte les contraintes de facilité d'utilisation, de généricité et d'extensibilité et les contraintes techniques.

2.2 Arbres de comportement

Nous voulons implémenter une méthode de simulation comportementale extensible, facile d'utilisation et surtout capable de produire, comme nous l'avons déjà énoncé, des agents autonomes adaptatifs scéranisés. Dans les paragraphes qui suivent nous allons décrire les origines, l'architecture et la structure de l'approche choisie, ensuite nous allons illustrer de manière théorique le fonctionnement des arbres de connaissance en soulignant au lecteur les avantages de ladite méthode et les raisons pour lesquelles nous l'avons adoptée. Finalement nous allons parler du contexte d'application et de test proposé en utilisant les concepts de Game Design, ainsi que de les outils techniques choisis.

2.2.1 Origine des arbres de comportement

Les arbres de comportement ont été initialement conçus comme un langage de modélisation utilisé pour décrire les cahiers des charges des projets industriels et commerciaux de grande échelle ; cependant, dans les dernières années ils sont devenus

une des méthodes les plus utilisé dans la simulation comportementale des personnages non joueurs [FPGMGM+09], [SGJ+11]. Dans l'industrie des jeux vidéo nous les trouvons dans des titres très connus comme *Halo 2*, *Halo 3* et *Spore 2*.

D'après l'académie et la recherche nous avons trouvé dans [SGJ+11] l'introduction de l'utilisation de paramètres dans les arbres de comportement, ils argumentent qu'en ajoutant cette extension dans le modèle de base, les arbres de comportements vont bénéficier d'avantages propres aux langages de programmation orienté objets et ainsi pourront être créés comme des fonctions avec des arguments. [FPGMGM+09] proposent d'inclure des capacités de planification en cours d'exécution trouvées dans les raisonnements à base de cas ; pour cela ils ont défini un nouveau type de nœud qui représente des requêtes dans les sens de bases de donnés ainsi que le concept de sous-arbres réutilisables, chaque sous-arbre ayant été conçu pour atteindre des buts spécifiques. Ils ont ajoutés des descriptions à chaque sous-arbres pour spécifier le but qu'ils accomplissent. Avec ces modifications, un concepteur inclura pendant la phase de conception des nœuds de requête dans les arbres de comportements de chaque personnage non joueur ; ces nœuds de requête vont récupérer le sous-arbre le plus approprié au contexte actuel du jeu.

De leur côté, [PSRGM+11] proposent de réduire les problèmes liés à l'utilisation de la planification à partir de cas dans les jeux de stratégie en temps réel en permettant aux experts d'ajouter des compétences en matière de décision sous la forme d'arbres de comportements. Nous trouvons aussi que [PNOB11] et [LBC10] ont proposé d'utiliser des techniques évolutionnistes sur des arbres de comportements afin de produire des personnages joueurs compétitifs pour les jeux DEFCON et Mario AI Benchmark. Enfin, [MKSB11] ont également proposé d'utiliser des arbres de comportements pour décrire le comportement d'une caméra intelligente.

2.2.2 Structure d'un arbre de comportement

Les arbres de comportements sont une simple structure de données qui synthétise les avantages des automates à états finis, des réseaux de tâches hiérarchiques et de l'exécution d'actions [FPGMGM+09]. Ils permettent d'organiser des comportements d'une façon hiérarchique descendante, c'est à dire, les comportements plus larges et complexes se trouvent en haut de l'arbre et ils sont décomposés en plusieurs sous-arbres contenant des comportements plus simples.

Le bloc principal utilisé pour la construction d'un arbre de comportement est une **tâche**. Une **tâche** peut exécuter une action très simple comme changer la valeur d'un attribut, tester une condition ou montrer une animation. Nous pouvons regrouper plusieurs tâches dans un sous-arbre afin de produire des actions plus complexes.

L'exécution d'une tâche ou d'un sous-arbre donne comme résultat un **état** parmi les trois possibles montrés dans la Table 2.1 :

Dans le modèle de base nous trouvons trois types de **tâches** : **actions**, **conditions** et **composites**.

1. **Actions :** ce sont des méthodes exécutées dans le monde virtuels. Dans certaines situations elles peuvent produire des changements dans l'état de l'agent ou

Succès	L'exécution est finie d'une manière satisfaisante.
En exécution	La tâche ou le sous-arbre n'a pas encore fini son exécution.
Échec	La tâche ou le sous-arbre a échoué. Par exemple, la condition testée était fausse.

TABLE 2.1 – Résultats possibles d'une exécution de tâche

de l'environnement. Quelques exemples d'actions sont : calculer une position, déterminer une distance, exécuter une animation, etc.

2. **Conditions :** elles évaluent des propriétés de l'agent et de l'environnement. Quelques exemples de conditions sont : tester si l'agent est proche du joueur, vérifier que l'agent est visible pour la caméra, etc.
3. **Composites : elles regroupent des actions et conditions afin d'exécuter des tâches de plus haut niveau. Le résultat de leur exécution dépend du résultat des tâches regroupées**

Les actions et les conditions représentent les feuilles de l'arbre, tandis que les composites représentent les nœuds. Ces derniers peuvent aussi être regroupés dans d'autres composites.

Comme dans le cas des tâches, il y a aussi trois types différents de composites(nœuds) : **nœuds de séquence**, **nœuds de sélection** et **nœuds parallèles**. La principale différence parmi ces composites est la façon dont ils parcourent les tâches ou les nœuds qui les forment.

1. **Nœuds de séquence :** ils représentent la façon plus simple d'assembler des comportements. Ce type de nœud exécute ses fils les uns après les autres. Lorsque tous les fils du nœuds réussissent, il sera considéré comme réussit ; par contre, lorsque un de ses fils échoue, le nœud finira son exécution avec un état d'échec qui sera retourné vers son nœud père. Nous pouvons observer ce fonctionnement graphiquement dans la Figure 2.1.

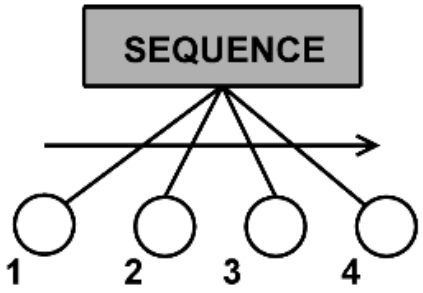

FIGURE 2.1 – Représentation graphique du nœud de séquence

2. **Nœuds de sélection :** ce sont le complément des nœuds de séquence. Au contraire de ces derniers, pour un nœud de sélection chaque nœud fils représente une alternative d'exécution. Lorsque un de ces nœuds fils réussit, il réussira aussi et lorsque un de ses fils échoue, le nœud de sélection continuera avec le prochain. Ce type de nœuds échouera seulement si tous ses fils ont échoué. Nous pouvons observer ce fonctionnement graphiquement dans la Figure 2.2.

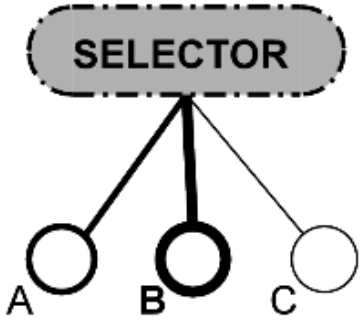

FIGURE 2.2 – Représentation graphique du nœud de sélection

3. **Nœuds parallèles :** ils exécutent tous leurs fils en simultané et par conséquent ils supportent des tâches concurrentes. Les politiques de succès et d'échec peuvent être définies selon les contextes d'application. Par exemple, nous pouvons considérer que ce type de nœud a échoué ou réussit si un nombre défini de ses fils ont échoué ou réussit aussi. La représentation de ce nœud se trouve dans la Figure 2.3.

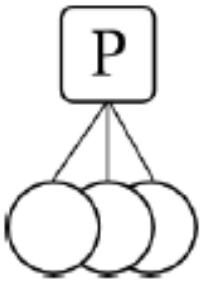

FIGURE 2.3 – Représentation graphique du nœud parallèle

Pour finir, nous trouvons un type de nœud particulier connu comme **nœud décorateur**. Il prend son nom du patron de conception *décorateur* proposé par le génie logiciel. Par définition, le patron *décorateur* fait référence à une classe qui enveloppe une autre, de telle façon que le comportement de cette dernière est modifié sans toucher sa définition initiale [BLKJ11]. En prenant cette idée, un **nœud décorateur** est un

nœud qui regroupe un seul et unique fils et modifie son exécution. Par exemple, nous pouvons définir un **nœud décorateur** qui exécutera son fils jusqu'à ce qu'il réussisse. La représentation de ce nœud se trouve dans la Figure 2.4.

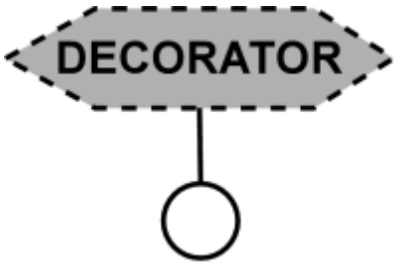

FIGURE 2.4 – Représentation graphique du nœud décorateur

2.2.3 Avantages des arbres de comportement

Une des principaux avantages des arbres de comportements est qu'ils ont la même puissance que les automates à état finis hiérarchiques sans souffrir des baisses de performance survenant lorsque la quantité des états et transitions augmentent au fur et à mesure que nous définissons de nouveau comportements. Les arbres de comportements permettent que les transitions vers des autres états soient auto-contenues [BLKJ11]. De la même façon, grâce à leur nature hiérarchique, ils peuvent évoluer facilement car la détermination d'un comportement se fait sur plusieurs niveaux d'abstraction [FPGMGM$^+$09].

Comme l'arbre est construit par regroupement de tâches simples, cela reste une solution assez rapide, générique, modulaire et facilement réutilisable. Un autre avantage est que, grâce à leur nature réactive, les agents seront capables de répondre de manière presque immédiate aux interventions du joueur en utilisant les comportements que le concepteur a défini dans son scénario. De plus, les arbres de comportement peuvent être facilement implémentés tant par les programmeurs que par les personnes avec peu au pas de connaissance en programmation car leur construction est incrémentielle ; ils sont aussi facilement contrôlés et testés [PSRGM$^+$11].

D'autre part, si la construction de chaque tâches reste assez générale, nous pourrons définir des comportements applicables dans un contexte quelconque. Cela nous permettra aussi de commencer l'intégration d'une base de comportements applicables aux nouveau projets de l'équipe Vortex ainsi que dans les projets existants. Pour tous ces avantages et du fait que jusqu'à aujourd'hui, selon la littérature, ils n'ont pas encore été utilisés dans les jeux sérieux nous avons décidé de choisir les arbres de comportements comme méthode de simulation comportementale.

2.3 Jeu sérieux : Prendre le métro, une question de civilité

Dans [AD10], les auteurs définissent un jeu sérieux comme : « *tout jeu intentionnellement conçu afin de viser une ou plusieurs des finalités définies et dont le segment de*

marché dépasse celui du seul divertissement ≫. En d'autres termes, un jeu sérieux est une application qui combine le jeu vidéo et une ou plusieurs fonctions utilitaires comme la diffusion d'un message, l'échange de données, l'entrainement ou l'apprentissage d'un bon comportement dans certaines situations. De la même façon, un jeu sérieux se différencie d'un jeu vidéo parce qu'il vise un marché autre que celui du seul divertissement ; nous trouvons des jeux sérieux utilisés par la défense, la formation, l'éducation, la santé, le commerce, la communication, etc.

D'autre part, l'équipe Vortex porte aujourd'hui un grand intérêt sur ce type d'applications et les avantages que leur utilisation peuvent apporter à la recherche sur la simulation comportementale. Pour cela, nous avons décidé de définir un petit et simple jeu sérieux qui nous permettra de tester la portée et les limites des arbres de comportement. Le marché choisi est celui de la formation, et dû aux des nombreux projets dans les équipes qui s'intéressent à la simulation des comportements observés dans la ville, nous avons proposé comme contexte et concept du jeu l'utilisation correcte du métro.

Pour la définition et conception de notre jeu, nous avons utilisé le modèle proposé par [Dja11]. Ce modèle reprend des concepts trouvés dans le ≪ Game Design ≫ et les applique à la conception et réalisation des jeux sérieux. Nous remarquons que le Game Design correspond selon l'auteur au processus global de création d'un jeu : définition de contenu, définition des niveaux, réalisation de prototypes, etc. Le modèle proposé est connu comme le modèle **DICE : Définir, Imaginer, Créer, Évaluer** où chaque étape correspond à :

- **Définir :** spécification du contenue sérieux (un message, un savoir faire, une bonne pratique, etc.) que nous voulons transmettre par le jeu.
- **Imaginer :** à partir de ce contenu sérieux le concepteur ou le créateur invente un concept de jeu.
- **Créer :** un prototype est réalisé pour tester la pertinence de ce concept de jeu.
- **Évaluer :** le prototype est évalué auprès d'un public cible. Nous voulons savoir si la transmission du contenu sérieux est effective.

En suivant ce modèle nous vous présentons dans la suite de ce document la définition de notre jeu ≪ **Prendre le métro, une question de civilité** ≫.

2.3.1 Définition du contenu sérieux

Le métro est une des solutions aux demandes de transport et mobilisation massive et rapide des citoyens les plus adoptées dans les villes françaises. Les villes comme Paris, Rennes, Lyon et Toulouse entre autres ont créé des réseaux de métro qui permettent de les traverser du nord au sud et de l'est à l'ouest en quelques minutes et à bas prix. On estime que dans une journée environ 149.916 voyeurs utilisent le métro à Toulouse, 565.900 à Lyon, et 4.050.000 à Paris.

Avec de telles quantités d'usagers du service par jour, plusieurs règles et conseils sont mis en œuvre afin de voyager en totale sécurité, qualité et commodité, ainsi que de faire une bonne utilisation du réseau de métro.

Dans ce contexte et en vue des différentes campagnes publicitaires qui cherchent à faire prendre conscience aux voyageurs des règles de comportement qui permettent de

bien utiliser le service en respectant toujours le consignes de sécurité, convivialité, civilité et solidarité, nous proposons la réalisation d'un jeu sérieux.

L'objectif de ce jeu est d'apprendre aux enfants ainsi que de rappeler aux voyageurs fréquents et non fréquents, la façon correcte de prendre le métro une fois qu'on se trouve dans le quai.

Après avoir visité les différents sites web des réseaux métro français, nous pouvons établir que les règles de comportements les plus générales et pertinentes à suivre dans les quais du métro sont :

1. Ne gênez pas la montée ou la progression des autres voyageurs en obstruant les couloirs et passages.
2. Les voyageurs peuvent monter par toutes les portes.
3. Laissez descendre les autres voyageurs avant de monter, afin de gagner du temps et d'éviter les bousculades. Placez-vous à côté des portes.
4. Il est demandé de faciliter l'accès et de donner un traitement préférentiel aux personnes à mobilité réduite, femmes enceintes, personnes âgées et personnes accompagnées de jeunes enfants.
5. N'entrez pas dans les véhicules ou n'en sortez pas pendant la fermeture des portes.
6. Ne gênez pas la fermeture des portes.
7. Ne courrez pas sur les quais.
8. Ne forcez pas les portes des rames.
9. Ne montez pas dans le métro et n'en descendez pas à partir du moment où retentit le signal sonore annonçant la fermeture des portes.

2.3.2 Imaginer le concept du jeu

En prenant en compte le contenu sérieux déjà présenté, nous proposons de réaliser un jeu où l'objectif du joueur est de réussir à monter dans une de voitures du métro pendant le temps d'attente établit en respectant les règles et recommandations déjà listées ainsi qu'en montrant un comportement le plus civilisé possible. Sous ce concept, il peut arriver que le joueur réussisse à monter dans le métro ; cependant si ses actions vont à l'encontre des normes de civilité et de convivialité ainsi que des règles d'utilisation du service, son score final sera diminué en fonction des infractions commises. Dans le cas où le score, après la déduction de pénalisations respectives, est inférieur à un certain seuil le jeu sera considéré comme perdu.

Pour mesurer si le joueur suit ou non les recommandations sur l'accès aux voitures du métro, ainsi que si son attitude et son comportement sont en concordance avec ce qu'on a définit comme un comportement civilisé dans les transports public et massifs, une seule tentative pour gagner n'est pas suffisante et s'éloigne de la réalité. Les comportements adéquats et inadéquats dont nous voulons faire prendre conscience au joueur surgissent avec une utilisation fréquente du métro. Comme il a été montré par la campagne « cher voisin du transport » mis en place par le système RATP de Paris, certains voyageurs font appel aux actions comme bousculer, ne pas laisser la place aux passagers prioritaires,

insulter, entre autres que lorsque qu'ils ont été déjà victimes des mêmes comportements « incivils ».

En prenant en compte ces observations, nous croyons qu'une des approches à utiliser pour mieux transmettre le contenu sérieux du jeu est de demander au joueur d'atteindre le même objectif plusieurs fois pendant la même partie du jeu. La différence entre la première fois et la deuxième et ainsi de suite réside dans la modification des comportements des personnages non joueurs ainsi que des changements dans l'environnement du jeu (un temps d'attente plus long, augmentation des personnages non joueurs sortants du métro, entre autres) par rapport à la performance et aux actions du joueur lors de la partie précédente.

Pour atteindre son but, le joueur pourra piloter un avatar qui se trouve dans un de quai du réseau métro de Toulouse. Les actions que l'avatar peut exécuter sont : marcher, courir, s'arrêter, bousculer les autres voyageurs et répondre aux actions des personnages non joueurs de la même façon qu'eux ont agi. D'autre part, pour monter dans le métro, le joueur dispose d'environ 30-40 secondes entre le moment où le métro s'arrête et ouvre ses portes jusqu'au moment où le signal de la fermeture des portes est déclenché. Pendant la durée de la partie pour atteindre son but, le joueur doit respecter les contraintes suivantes :

- Ne rentrez pas dans les véhicules du métro ou n'en sortez pas à partir du moment où le signal de fermeture de portes est annoncé (cela implique aussi pendant la fermeture de portes).
- Placez-vous à côté des portes/rames, et laissez descendre les autres voyageurs avant de monter.
- Ne bousculez pas les autres voyageurs afin d'entrer ou de sortir du métro. (Civilité)
- Facilitez l'accès aux personnes à mobilité réduite, femmes enceintes, personnes âgées et personnes accompagnées de jeunes enfants ; donnez-leur un traitement préférentiel (laissez les places assises, etc.) (Civilité)
- Ne vous placez pas devant les portes du métro jusqu'à que votre station de destination soit proche.
- Ne courrez pas sur les quais.
- Ne gênez pas la fermeture ni l'ouverture des portes (inclût forcer les portes).

Les contraintes listées sont liées aux règles et recommandations du service métro, ainsi qu'à la civilité.

Pour quantifier la performance du joueur à chaque partie, nous allons mesurer : les nombre de fois où le joueur réussit à prendre le métro, le temps utilisé pour monter dans les voitures du métro, la quantité des recommandations et règles respectées et l'état émotionnel final des personnages non joueurs à chaque itération d'une partie.

2.3.3 Sélection d'une usine à jeux : Unity3D

Lors de la conception d'un jeu sérieux ou d'un jeu vidéo, nous pouvons nous servir des outils théoriques (comme le modèle DICE) et des outils techniques (appelés dans les cas des jeux : usines à jeux). Nous avons déjà un concept de jeu, nous devons ensuite trouver les outils nous permettant de créer des prototypes et de tester notre concept. Dans son travail de thèse, [Dja11] a réalise une classification des 363 usines à jeux permettant

la création d'un genre de jeu en particulier (aventure, combat, stratégie) ainsi que la création de tout type de jeux vidéo.

Pour choisir notre outils, nous avons repris cette classification et nous avons choisi les outils permettant de créer tout type de jeux vidéo, particulièrement ceux capables de produire un résultat graphique en 3D. L'application de ce filtre nous a donné 46 usines. Parmi celles-ci, nous avons choisi celles qui peuvent être acquises de manière gratuite, avec un bon support technique et une communauté très active, ainsi que capables d'exécuter le code produit par un langage de programmation commun ou avec un langage de script propriétaire puissant. Avec ces nouveaux critères, les usines à jeux résultantes sont listées dans la Table 2.2.

Outils	**O.S**	**Langage**	**Licence**	**Année**
3D Rad 6	Windows	Script propriétaire	Freeware	2009
Antiryad GX	Windows/ Linux/Mac	Script propriétaire/ langage commun	Commercial/ Free	2003
Crystal Architect	Windows/ Linux/Mac	Langage commun	Commercial/ Open source	2011
Dimension3	Mac	Script propriétaire/ langage commun	Commercial/ Free	2010
DX Studio 3.0	Windows	Langage commun	Free	2008
Original3D : Game Creator	Windows	Langage commun	Free	2007
Ray Game Designer 2	Windows	Script propriétaire	Free	2002
RayDrame : Game Studio	Windows/ Linux/Mac	Script propriétaire	Open source	2006
Reality Factory	Windows	Script propriétaire	Open source	2008
Sandbox 3D Game Maker	Windows/ Linux/Mac	Script propriétaire	Open source	2009
ShiVa	Windows	Langage commun	Commercial/ Free	2007
Unity3D 2.6	Windows/ Mac	Script propriétaire/ langage commun	Commercial/ Free	2009
Unreal Development Kit	Windows/ iOS	Langage commun	Commercial/ Free	2009

TABLE 2.2 – Usines à jeux résultants

Parmi ces 13 usines à jeux nous avons décidé de choisir **Unity3D**, car il supporte un langage de programmation *C#* avec lequel nous avons déjà travaillé. La communauté des concepteur et créateur de jeu est assez large, cela nous facilitera le processus d'apprentissage de l'outil. Des plus, l'équipe Vortex compte plusieurs membres qui travaillent actuellement sur cet outil et par conséquent nous aurons à notre disposition leurs expériences et connaissances sur Unity3D. Finalement, nous trouvons qu'en prenant

en compte nos contraintes du temps, Unity3D est à la fois assez simple pour nous permettre de développer de façon très rapide et facile notre jeu, et assez puissant pour supporter l'implémentation des méthodes de simulation comportementale choisi. Les Figure 2.5 et 2.6 montrent l'environnement virtuel 3D que nous avons construit en utilisant cet outil.

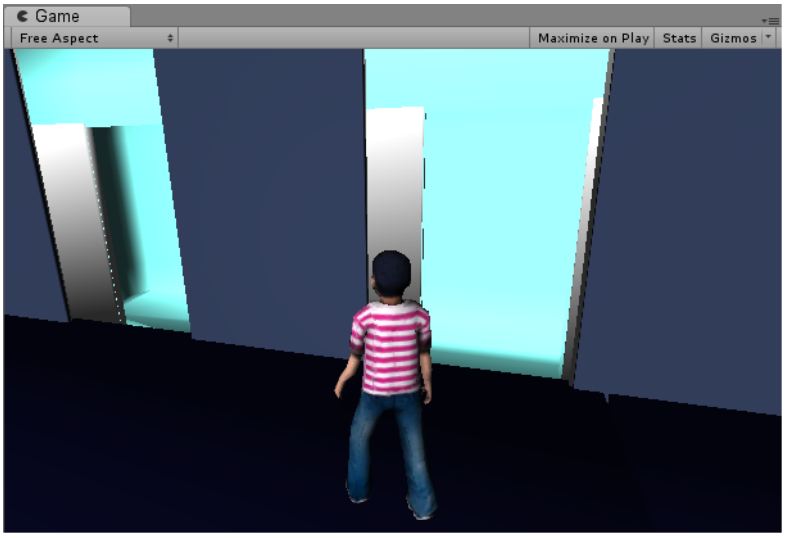

FIGURE 2.5 – Modèle 3D avec le métro arrêté

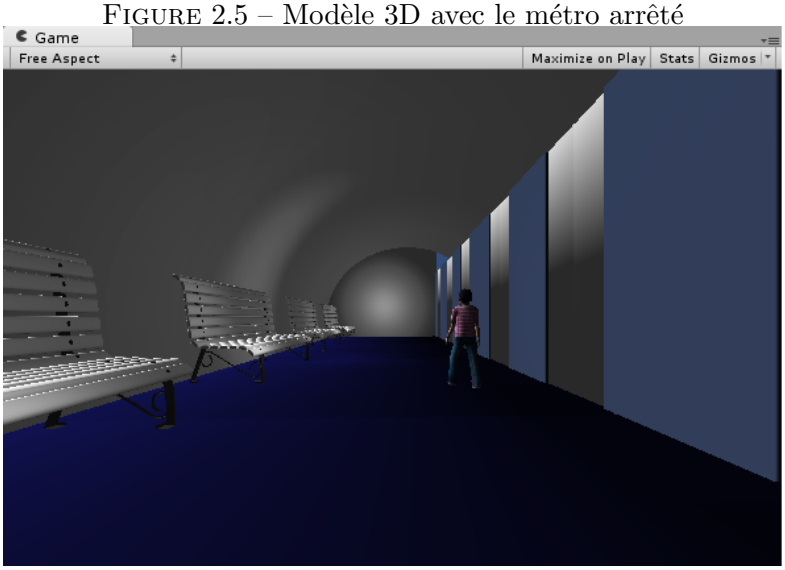

FIGURE 2.6 – Modèle 3D du quai métro

2.4 Implémentation des arbres de comportements et intégration avec notre jeu

A ce stade, nous avons choisi une méthode, un outil et un contexte d'application. Dans la section suivante nous décrirons en détail la façon dont les arbres de comportement ont été implémentés sur Unity3D, ainsi que le mécanisme utilisé pour la création de nos agents autonomes adaptatif scénarisés. Finalement, nous expliquerons l'architecture de nos agents et son intégration avec les arbres de comportements. Cependant, avant de rentrer dans tous ces détails, nous énoncerons quelques spécificités propres à Unity3D qui ont été très importantes pour le développement de notre jeu et desquelles nous allons nous servir justifier notre démarche.

2.4.1 Quelques clarifications sur Unity3D

Le bloc de construction de base de tout jeu développé à travers Unity3D est un **Game Object** ; chaque objet dans notre jeu (un personnage non joueur, le métro, la caméra principale, etc.) est un Game Object. Ce qui fait la différence entre un GameObject et un autre sont les **Components** (composants) qui l'intègrent, c'est à dire, selon le type d'objet que nous voulons créer nous aurons une combinaison des composants différents. Unity3D nous offre des composants prédéfinis comme les **transformations** qui définissent la position, l'échelle, etc. de notre objet ; ou **colliders** et **rigidbodies** qui permettent à notre object d'être affecté par la physique, etc.

Si nous souhaitons que nos objets se comportent d'une façon particulière, nous devons ajouter un composant de type **Script**. Un script est un morceau du code implémenté en utilisant les langages supportés par Unity3D (C# , JavaScript ou Boo) qui sera traité et exécuté par Unity3D comme tout autre composant. Par définition, un script dont nous souhaitons qu'il soit exécuté par le moteur du jeu (Unity3D) doit hériter de la classe **MonoBehavior** définie par Unity3D. Cette classe mettra à notre disposition des méthodes qui seront appelées à chaque frame (*Update, FixedUpdate et LateUpdate*), des méthodes qui seront appelées une seule fois pendant que l'objet est chargé en memoire (*Awake, Start*) et des événements qui vont nous informer sur un quelconque changement dans l'environnement comme par exemple une collision [Tec12].

Par conséquent, dans les cas où nous souhaitons qu'un objet réalise certaines action lorsqu'une certaine condition est vraie (comme déterminer si la distance entre notre personnage et le joueur est inférieure à un seuil défini afin de déclencher une action), les contraintes d'exécution nous obligent à tester cette condition dans une des méthodes d'actualisation et comme ces méthodes sont appelées à chaque frame (environ entre 30-40 fois par seconde) notre code réalise du travail inutile et nous gaspillons des ressources de calculs. Ce phénomène est aggravé lorsque nous souhaitons exécuter un processus ou une action dense qui se déroule pendant de multiple frames, car nous sommes donc obligé de le faire en un seul moment, même à des instants où il n'est pas nécessaire de le faire.

La solution naturelle à ce problème est de diviser le travail de notre processus en petits morceaux qui seront exécutés dans des frames différents et où chaque morceau correspond à une action en particulière. Cependant, l'exécution et coordination de chaque morceau

résultant, dans le cas de processus de nature longue et diverse, est très complexe et fastidieuse. Pour résoudre ce problème Unity3D nous offres les **Coroutines** [Fin11].

2.4.2 Notion de coroutine

Intuitivement, une coroutine est un type d'application qui nous permet d'arrêter son exécution à un endroit donné et de la reprendre au même endroit lorsque nous en avons besoin ou lorsque certaines conditions sont vrai. Dans le cas d'Unity3D, une coroutine nous permet de répartir l'exécution de l'ensemble de lignes de code qui forment un processus sur plusieurs frames.

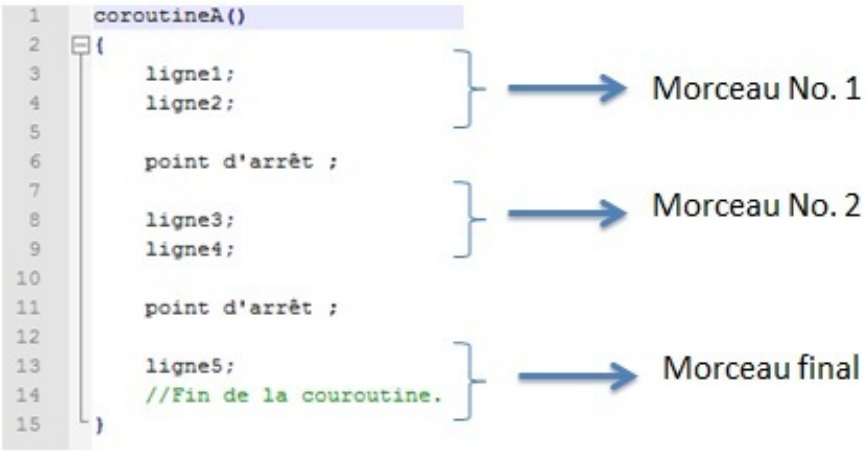

FIGURE 2.7 – Pseudo-code d'une coroutine

Nous allons utiliser la Figure 2.7 pour mieux expliquer le concept de coroutine. Supposons que la **coroutineA()** illustrée dans la Figure 2.7 est appelée par la méthode **Update()** à chaque frame. Par définition, l'exécution se passera ainsi :

1. Lors du premier appel, les lignes de code du **Morceau 1** seront exécutées, cependant grâce au **point d'arrêt** l'exécution de la **coroutineA()** sera arrêtée et le contrôle retournera à la fonction **Update()**.
2. Comme la fonction **Update()** est exécutée en entier à chaque frame, elle fera appel encore une fois à la **coroutineA()** ; cependant selon sa définition de coroutine, elle reprendra son exécution juste après le premier **point d'arrêt**, en d'autre termes, elle exécutera les lignes de code du **Morceau 2** et stoppera son exécution encore une fois grâce au deuxième **point d'arrêt**.
3. Ce fonctionnement continuera jusqu'à atteindre la fin de la coroutine. Lorsque les appels depuis la fonction **Update()** atteindront la fin de la **coroutineA()**, celle dernière recommencera son exécution de la même façon.

Ce principe est implémenté en utilisant les fonctions **IEnumerator** et le mot clé **yield**. Nous pouvons trouver un exemple dans la Figure 2.8.

```
IEnumerator TellMeASecret()
{
  PlayAnimation("LeanInConspiratorially");
  while(playingAnimation)
    yield return null;

  Say("I stole the cookie from the cookie jar!");
  while(speaking)
    yield return null;

  PlayAnimation("LeanOutRelieved");
  while(playingAnimation)
    yield return null;
}
```

FIGURE 2.8 – Coroutine écrite en Csharp

La fonction **IEnumerator** travaille comme un pointeur sur une séquence et met à notre disposition deux membres importants : **Current** et **MoveNext()**. Le premier est une propriété qui nous donnera toujours l'élément sur lequel se trouve actuellement le pointeur ; le deuxième est une fonction qui nous donnera l'élément suivant dans la séquence. La fonction **IEnumerator** est une interface, cela implique que normalement si nous voulons l'utiliser nous devons définir une classe qui l'implémente ; heureusement en utilisant les blocs itérateurs et en respectant certaines règles ce sera le compilateur qui générera notre implémentation [Fin11].

Un bloc itérateur est une fonction qui retourne une variable de type **IEnumerator** et utilise le mot clé **yield**. Le mot clé **yield** nous permet de déclarer ce que nous voulons être le prochain élément de la séquence ou de signifier qu'il n'y a plus d'éléments à parcourir. Lorsque le code rencontre un **yield return X**, l'exécution de la fonction **MoveNext()** sera stoppée, la fonction retournera **true** et **Current** prendra la valeur X. Cela implique que lorsque nous appelons la fonction **MoveNext()**, le bloc itérateur va se déplacer vers le point où est déclaré le prochain **yield** et nous aurons l'exécution en morceaux par frame que nous avons décrite dans les paragraphes précédents [Fin11].

Afin d'optimiser le fonctionnement de notre arbre de comportements et de l'intégrer de la meilleur façon possible avec l'architecture et les comportement propres à Unity3D nous avons décidé d'utiliser la notion de coroutine pour son implémentation.

2.4.3 Notre implémentation des arbres de comportements

La Figure 2.9 montre l'architecture de classes que nous avons utilisé pour implémenter nos arbres de comportement. Tout d'abord nous avons défini une énumération appelée **BehaviorResult** qui contient les trois résultats possible lorsque notre système exécute une tâche ou un comportement.

Par la suite, nous avons défini une interface appelée **IBehavior** qui contient comme

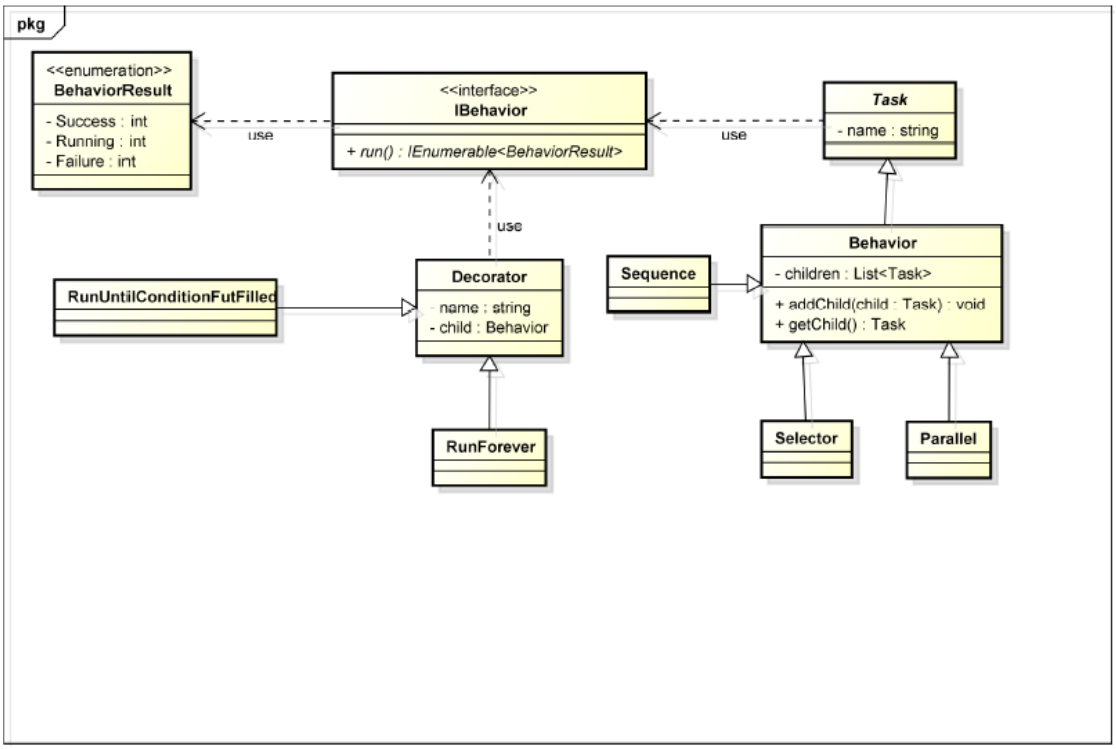

FIGURE 2.9 – Diagramme de classe d'arbres de comportement

membre une unique fonction de type **IEnumerable**, car au lieu d'accéder à une seul itération comme le fait **IEnumerator** cette interface nous donne un accès direct et complet au bloc d'itérateurs. Comme vous pouvez regarder dans la Figure 2.9, l'interface **IBehavior** et son unique méthode **Run()** sont la base de notre implémentation. Dans cette méthode **Run()** doit être décrit tout le comportement d'une tâche (soit une action, condition ou composite) ainsi que le comportement d'un décorateur. Par exemple, si nous avons un tâche **calculateDistanceBetweenPoints** c'est dans la méthode **Run()** que nous définirons la logique de calcul ; pour notre exemple nous calculerons la distance euclidienne entre deux points. Il est important de remarquer que ce que retourne cette méthode nous indiquera si l'exécution est un succès ou un échec, ou si elle continue encore.

Nous avons défini aussi deux classes de base **Task** et **Decorator**. Celles-ci implémentent la méthode **Run()** et serviront comme classes mères pour des tâches, des comportements et des décorateurs. De plus, la classe **Behavior** est utilisée comme modèle générique de nos nœuds ed séquence, sélection et parallèle ; pour cela elle inclut comme paramètre la liste de tâches regroupées en chaque nœud. Finalement, nous faisons remarquer que l'unique différence entre nos classes **Selection**, **Sequence** et **Parallel** réside dans la façon dont elle parcourent leurs fils.

Pour illustration, nous présenterons l'implémentation de la méthode **Run** que nous trouverons dans une classe **Sequence**.

```
override public IEnumerable<BehaviorResult> Run()
{
 foreach(Action child in children)
  {
   foreach(BehaviorResult result in child.Run())
        {
          switch(result)
                {
                  case BehaviorResult.Success:
                        Debug.Log ("child executed successfully");
                        goto NextChild;
                  case BehaviorResult.Failure:
                        yield return BehaviorResult.Failure;
                        yield break;
                 case BehaviorResult.Running:
                        yield return BehaviorResult.Running;
                        continue;
                }
        }
        NextChild:;
  }
 yield return BehaviorResult.Success;
 yield break;
}
```

2.4.4 Mécanisme de création des agents autonomes

Dans Unity3D, un **Prefab**, ou préfabriqué en français, est une collection de GameObjects ou de composants qui peuvent être réutilisés dans notre environnement virtuel. Nous nous servons de cette fonctionnalité, lors de la création de nos personnages non joueurs. Pour cela, nous avons défini une nouvelle classe **Director**, héritière de de la classe **MonoBehavior** créée par Unity3D, qui est en charge de récupérer les spécifications -au sujet de la création des personnages non joueur- fournies par le créateur du jeu et de les inclure dans l'environnement virtuel. Ces spécifications sont transmises sous la forme d'un fichier XML, la Figure 2.10 est un exemple du contenu de ce fichier.

Pour chaque personnage non joueur (PNJ) le créateur doit préciser : un nom ; trois valeur dans une échelle de 0 à 10 pour représenter trois émotions du personnage : son niveau de hâte, un niveau de tolérance aux incivilités et un niveau de sociabilité ; une position, une rotation et une liste de comportements pour construire son arbre de comportements. Pour créer chaque comportement le créateur doit définir un nom qui équivaut aux comportements ou actions déjà préprogrammées, une durée pour l'exécution du comportement, une destination dans le cas des comportements qui impliquent un déplacement et un type si nous souhaitons que l'arbre contienne un nœud de sélection qui regroupe les comportements du même type.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!-- This document was created with Syntext Serna Free. --><characters>
  <Character name="Alex" hurried="2" tolerance="5" sociable="6">
    <Position X="58" Y="5.37" Z="-27"/>
    <Rotation X="0" Y="90" Z="0"/>
    <Behavior name="WalkTo" destination="door"/>
    <Behavior name="Talk"  type= "Wait" duration="10"/>
    <Behavior name="Read" type= "Wait" duration="10"/>
    <Behavior name="WalkTo" destination="center"/>
    <Behavior name="TakeMetro" destination="insideMetro"/>
  </Character>
  <Character name="Katty" hurried="5" tolerance="8" sociable="4">
    <Position X="80" Y="5.38" Z="146"/>
    <Rotation X="0" Y="-90" Z="0"/>
    <Behavior name="WalkTo" destination="door"/>
    <Behavior name="TakeMetro" destination="insideMetro"/>
  </Character>
</characters>
```

FIGURE 2.10 – Exemple fichier XML

Si nous regardons la première entrée du type *Character* dans la Figure 2.10, la création du personnage se fera de la façon suivante :

1. La classe Directeur réalisera une instance d'un préfabriqué qui contient les composants de base pour notre personnage. Nous donnerons plus de détail sur l'architecture du personnage dans la sous-section suivante.
2. Le préfabriqué aura comme attributs initiaux les valeurs définies dans les champs *name*, *hurried*, *tolerance* et *sociable* ; en effet, nous aurons un personnage nommé Alex qui aura comme émotions un niveau de hâte égal à 2, un niveau de tolérance de 5 et un niveau de sociabilité égal à 6.
3. Ensuite, notre classe Directeur créera l'arbre de comportement qui aura comme racine un nœud de séquence formé par un nœud parallèle appelé **WalkTo**, un nœud de sélection appelé **Wait** qui comprend deux nœud de séquence **Talk** et **Read** et un autre nœud parallèle **WalkTo**. L'arbre finira avec un nœud de séquence appelé **TakeMetro**. Il est important de remarquer que les nœuds que nous venons de définir contiennent d'autres comportements et tâches et nous avons seulement donné un aperçu de l'arbre que nous considérons suffisant pour comprendre le processus décrit. La Figure 2.11 montre l'arbre résultant.

2.4.5 Architecture finale des PNJ

Comme nous l'avons déjà énoncé la création d'un objet en Unity3D, quelque soit sa finalité (un personnage, une voiture, etc.), se fait par composition. Dans la suite nous présenterons les différents composants qui intègrent nos personnages non joueurs et la façon dont ils interagissent avec nos arbres de comportements.

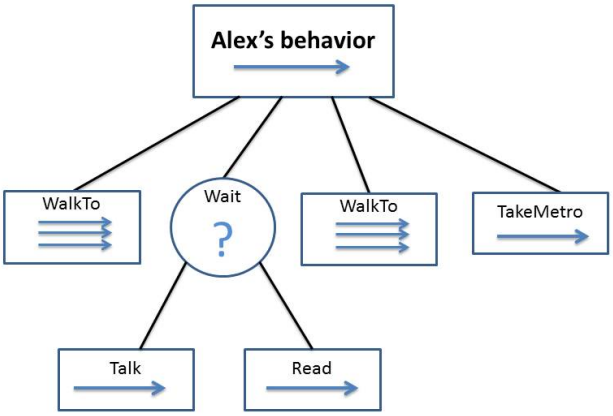

FIGURE 2.11 – Exemple d'arbre de comportement crée par notre système

Afin que notre personnage soit capable d'exécuter les comportements décrits par le créateur en restant toujours autonomes, adaptatif et réactive, nous avons conçu l'architecture présentée dans la Figure 2.12.

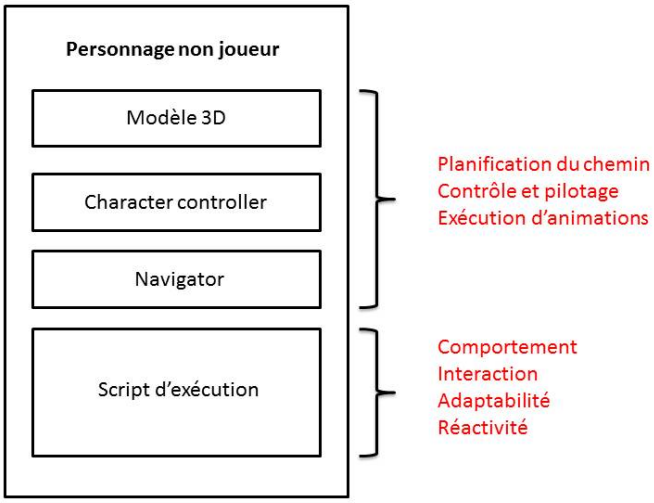

FIGURE 2.12 – Architecture d'un personnage non joueur

Dans cette architecture nous pouvons distinguer deux blocs principaux : **a.)** un bloc composé par un **modèle 3D** à travers lequel notre personnage est incarné illustré dans la Figure 2.13 ; un **navigateur** qui lui permet de définir et planifier un chemin optimal et libre d'obstacles statiques lorsque son arbre de comportement lui demande de se déplacer vers une destination déterminée ; et un **character controller**, illustré dans la Figure 2.14, qui sert comme interface entre notre arbre et Unity3D. Des tâches telles que : exécuter une animation, changer de position, tester une collision, entre autres se font à travers les méthodes de ce composant. Ce composant reçoit aussi toutes les notifications d'interaction avec l'environnement et avec le joueur, donc il est la base de la réactivité et de l'adaptabilité de notre personnage. Avec ce bloc nous résolvons les problèmes liés à la recherche et à la planification du chemin ainsi que ceux liées au contrôle et pilotage du mouvement décrits dans l'état de l'art.

FIGURE 2.13 – Modèle 3D du personnage

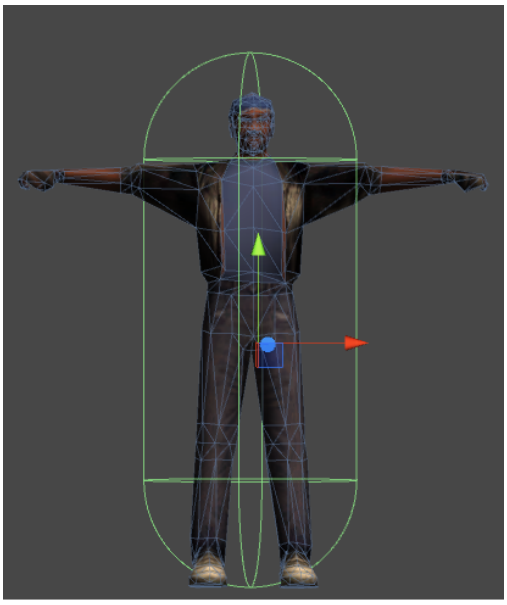

FIGURE 2.14 – Personnage avec un character controller

Et **b.)** un bloc composé par une classe appelée **Controller**, héritière de la la classe **Monobehavior** qui fait le lien entre Unity3D et notre arbre de comportement. Grâce à cette classe nous pouvons faire appel à notre arbre de comportement quand nous en avons besoin ou quand un certain changement est survenu. Cette classe sert aussi comme lien entre notre arbre et les autres composants définis dans le premier bloc. Par exemple, si nous reprenons l'arbre de la Figure 2.11, lorsque le comportement **WalkTo** commence à être exécuté, cette classe fera le lien entre le **navigateur** qui nous donnera le chemin, le **character controller** qui déplacera le **modèle 3D** tout au long du chemin reçu et qui exécutera une animation de marche et l'arbre de comportement qui dicte dans quel ordre et de quelle façon doivent s'exécuter ses tâches. C'est aussi cette classe qui lancera le comportement **Wait** si le comportement **WalkTo** est fini avec succès.

Comme dernière remarque, nous avons décidé d'inclure dans cette classe les trois

émotions de hâte, tolérance et sociabilité comme un moyen de montrer la versatilité de nos arbres. Par exemple, en prenant en compte l'arbre de la Figure 2.11 et en particulier le nœud de sélection **Wait** c'est le niveau de sociabilité qui conditionnera le choix de notre personnage entre le comportement de **Read** et **Talk**. Ainsi, le niveau de hâte déterminera aussi la façon dont notre personnage se déplacera, car un personnage pressé marchera plus rapidement et fera moins attention aux autres. Cette fonctionnalité nous permet de définir des personnages avec des arbres de comportements identiques mais avec des comportements finaux différents.

Conclusion

Dans ce document nous avons proposé l'utilisation d'arbres de comportement comme solution à la problématique de simulation comportementale et de sélection d'actions chez les personnages non joueurs qui peuplent un monde virtuel. Nous avons décider de choisir cette méthode, parce qu'elle est déjà utilisée dans l'industrie du jeu vidéo pour son efficacité et sa facilité d'utilisation ; elle résout des problèmes posés par d'autres approches comme les automates à état finis, les arbres de comportements, etc. ; elle n'a jamais été utilisé pour les jeux sérieux et parce que les résultats scientifiques publiés ces trois dernières années montrent la versatilité et l'extensibilité de cette approche. Pour l'instant nos personnages sont équipés de comportements réactifs simples formés en regroupant, selon le scénario défini, des tâches que nous avons préalablement programmées ; cependant comme il a été montré dans les travaux menés par [PNOB11] et [LBC10], nous pouvons facilement étendre les arbres de comportement afin de produire des enchaînements d'actions et de comportements délibérés.

La méthode que nous proposons pour construire des personnages à partir d'un fichier XML demande le travail conjoint du programmeur et du concepteur, car le concepteur ne pourra pas créer un comportement si les tâches de bases ne sont pas encore codées ; pour dépasser cette limitation nous proposons comme travail futur la construction d'une base de données commune qui contiendra des tâches génériques ou facilement paramétrables qui pourront être utilisées et partagées par plusieurs projets de l'équipe Vortex.

Par ailleurs, maîtriser Unity3D nous a pris plus de temps que ce que nous avions prévu et par conséquent nous n'avons pas pu tester, auprès d'utilisateurs, la crédibilité et le réalisme de nos agents ainsi que leurs contributions au sentiment de **flow** et de compréhension du contenu sérieux. Pour évaluer les arbres de comportement, nous proposons de distribuer le jeu à travers les sites web des différentes entreprises gestionnaires du service métro en France et de mesurer en utilisant des questionnaires courts quelles sont les impressions, les opinions et les recommandations des utilisateurs.

Finalement comme dernière amélioration, nous proposons de développer un outil graphique qui permettra la construction d'arbres de comportement à partir d'interactions comme : drag-and-drop, édition via des menus, sélection de paramètres. Cet outil facilitera encore plus le travail des concepteurs novices en programmation.

Bibliographie

[AD10] Julian Alvarez and Damien Djaouti. *Introduction au Serious Games*. Ludoscience, 2010.

[And03] Eike F. Anderson. Playing smart - artificial intelligence in computer games. In *Proceedings of zfxCON03 conference on game development*, 2003.

[BKKP06] Stephen Burman, Yang Sok Kim, Byeong Ho Kang, and Gil-Cheol Park. Maintenance of game character's AI by players. *International Journal of Multimedia and Ubiquitous Engineering*, 1(1), 2006.

[BLCR09] Jean-Marie Burkhardt, Domitile Lourdeaux, Stanislas Couix, and Michael Rouillé. La modélisation de l'activité humaine finalisée. In *Le traité de la réalité virtuelle : Les humains virtuels*, pages 213–243. Presses de Mines, 2009.

[BLKJ11] Iva Bojic, Tomislav Lipic, Mario Kusek, and Gordan Jezic. Extending the jade agent behaviour model with jbehaviourtrees framework. In *Agent and Multi-Agent Systems : Technologies and Applications*, volume 6682 of *Lecture Notes in Computer Science*, pages 159–168. Springer Berlin / Heidelberg, 2011.

[Bry01] Joanna J. Bryson. *Intelligence by Design : Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA, June 2001. AI Technical Report 2001-003.

[Bur06] Jean-Marie Burkhardt. Ergonomie, facteurs humains et réalité virtuelle. In *Le traité de la réalité virtuelle : L'homme et l'environnement virtuel*, pages 117–150. École de Mines de Paris, 2006.

[Cav00] M. Cavazza. Al in computer games : Survey and perspectives. *Virtual Reality*, 5 :223–235, 2000.

[Che07] Jenova Chen. Flow in games (and everything else). *Commun. ACM*, 50(4) :31–34, April 2007.

[Dja11] Damien Djaouti. *Serious GameDesign : Considérations théoriques et techniques sur la création de jeux vidéo à vocation utilitaire*. PhD thesis, Université Toulouse III Paul Sabatier, Novembre 2011.

[Don04] Stéphane Donikian. *Modélisation, contrôle et animation d'agents virtuels autonomes évoluant dans des environnements informés et structurés*. Habilitation à diriger des recherches, University of Rennes 1, 2004.

[DPP09] Stéphane Donikian, Nicolas Pépin, and Paolo Petta. D4.1 : State of the art report and requirement specifications. Technical report, INRIA, 2009.

[dS06] Etienne de Sevin. *An action selection architecture for autonomous virtual humans in persistent worlds*. Thèse de doctorat, École polytechnique fédérale de Lausanne, March 2006.

[Fin11] Richard Fine. Unity3d coroutines in detail, 2011. `http://www.altdevblogaday.com/2011/07/07/unity3d-coroutines-in-detail/`, consulté le 1/06/12.

[Flo03] Răzvan V. Florian. Autonomous artificial intelligent agents. Technical report, Coneural - Center for Cognitive and Neural Studies, February 2003.

[FPGMGM+09] Gonzalo Flórez-Puga, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, Belén Díaz-Agudelo, and Pedro Antonio González-Calero. Query-enabled behavior trees. *IEEE Transactions on computacional intelligence and AI games*, 1(4) :298–308, 2009.

[Isl05] Damian Isla. Handling complexity in the Halo 2 AI. In *Proceedings of the Game Developers Conference*, 2005.

[JLC05] Randolph M. Jones, Christian Lebiere, and Jacob A. Crossman. Comparing modeling idioms in ACT-R and soar. *Eighth International Conference on Cognitive Modeling.*, pages 49–54, 2005.

[JW01] Daniel Johnson and Janet Wiles. Computer games with intelligence. In *IEEE International Fuzzy Systems Conference*, pages 1335–1358, 2001.

[LBC10] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game defcon. In *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 100–110. Springer Berlin / Heidelberg, 2010.

[LD09] Fabrice Lamarche and Stéphane Donikian. La sélection d'action. In *Le traité de la réalité virtuelle : Les humains virtuels*, pages 259–283. Presses de Mines, 2009.

[LvL01] John E. Laird and Michael van Lent. Human-level AI's killer application : interactive computer games. *AI Magazine*, 22(2) :15–26, 2001.

[Mae94] Pattie Maes. Modeling adaptive autonomous agents. *Artifial Life*, Vol. 1(No. 1–2) :135–162, Fall 1993/Winter 1994.

[Mal97] Hanspeter A. Mallot. Behavior-oriented approachs to cognition : theorical perspectives. *Theory in biosciences*, pages 196–220, 1997.

[MF06] Daniel Mestre and Philippe Fuchs. Immersion et présence. In *Le traité de la réalité virtuelle : L'homme et l'environnement virtuel*, pages 309–338. École de Mines de Paris, 2006.

[MGVdS06] Jean-Sébastien Monzani, Anthony Guye-Vuilleme, and Etienne de Sevin. Behavioral animation. In *Handbook of Virtual Humans*, pages 260–285. John Wiley, 2006.

[MHK99] R.J Millar, J.R.P. Hanna, and S.M. Kealy. A review of behavioural animation. *Computers & Graphics : An International Journal of Systems & Applications in Computer Graphics.*, 23(Issue 1) :127–143, 1999.

[Mil06] Ian Millington. *Artificial intelligence for games*. Elsevier, 2006.

[MKSB11] Daniel Markowitz, Joseph Kider, Alexander Shoulson, and Norman Badler. Intelligent camera control using behavior trees. In *Motion in Games*, volume 7060 of *Lecture Notes in Computer Science*, pages 156–167. Springer Berlin / Heidelberg, 2011.

[MTT04] N. Magnenat-Thalmann and D. Thalmann. *An Overview of Virtual Humans*, pages 1–25. John Wiley, 2004.

[Nar04] Alexander Nareyek. AI in computer games. *Queue*, 1(10) :58–65, February 2004.

[PNOB11] Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 123–132. Springer Berlin / Heidelberg, 2011.

[PSRGM+11] Ricardo Palma, Antonio Sánchez-Ruiz, Marco Gómez-Martín, Pedro Gómez-Martín, and Pedro González-Calero. Combining expert knowledge and learning from demonstration in real-time strategy games. In *Case-Based Reasoning Research and Development*, volume 6880 of *Lecture Notes in Computer Science*, pages 181–195. Springer Berlin / Heidelberg, 2011.

[Rey99] Craig Reynold. Steering behaviors for autonomous characters. In *Proc. of Game Developers Conference*, pages 763–782, San Francisco, CA, 1999. Miller Freeman Game Group.

[SGJ+11] Alexander Shoulson, Francisco Garcia, Matthew Jones, Robert Mead, and Norman Badler. Parameterizing behavior trees. In *Motion in Games*, volume 7060 of *Lecture Notes in Computer Science*, pages 144–155. Springer Berlin / Heidelberg, 2011.

[Tec12] Unity Technologies. Creating gameplay, 2012. `http://unity3d.com/support/documentation/Manual/Creating%20Gameplay.html`, consulté le 1/06/12.

[Ver06] Jean-Louis Vercher. La latence temporelle dans la bucle de réalité virtuelle. In *Le traité de la réalité virtuelle : L'homme et l'environnement virtuel*, pages 340–355. École de Mines de Paris, 2006.

[Vey09] Morgan Veyret. *Un guide virtuel autonome pour la description d'un environnement réel dynamique. – Interaction entre la perception et la prise de décision.* PhD thesis, Université de Bretagne Occidentale, Laboratoire d'Informatique des Systèmes Complexes – EA3883, March 2009.