Ministère de l'Education Nationale

Université de Montpellier II

Rapport de Projet Informatique
du Master IFPRU 1ère année

Année 2008/2009

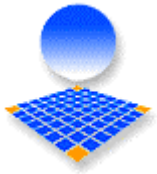
# Prise de décision dans les shooters

**Présenté et soutenu par :**

Romain ALMES
Sandrine BUENDIA
Coralie GALLIEN
Romain RICHARD

**Dirigé et encadré par :**

M. KORICHE



Ministère de l'Education Nationale

Université de Montpellier II

Rapport de Projet Informatique
du Master IFPRU 1ère année

Année 2008/2009

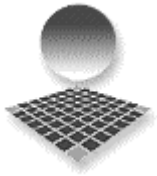
# Prise de décision dans les shooters

**Présenté et soutenu par :**

Romain ALMES
Sandrine BUENDIA
Coralie GALLIEN
Romain RICHARD

**Dirigé et encadré par :**

M. KORICHE



## Résumé

A l’heure où les graphismes s’approchent du photoréalisme et où les systèmes de jeux deviennent de plus en plus complexes, l’Intelligence Artificielle doit elle aussi progresser parallèlement, afin de donner l'illusion que les personnages non joueur ont une attitude se rapprochant de celle d’un joueur humain.

Dans ce contexte, nous avons implémenté le comportement de personnages non joueurs dans un jeu de type shooter (jeu de tir en vue subjective) grâce à un planificateur. La planification, qui consiste à sélectionner et à ordonnancer des actions permettant d’atteindre un but donné à partir d’une base de connaissances sur les actions possibles, offre une résolution des problèmes de façon dynamique, ce qui peut se révéler être un élément clé chez des personnages qui semblent comprendre leur environnement et réagir de façon logique.

## Remerciements

Pour la réalisation de ce projet, nous tenions à remercier nos professeurs, qui nous ont encadrés tout au long de cette année, et grâce à qui nous avons pu mener à bien le travail qui nous a été confié, depuis l'analyse jusqu'à la programmation.

Dans cette optique, nous remercions tout particulièrement notre tuteur de TER, M. KORICHE pour nous avoir encadrés et conseillés au long de ce projet.

## Sommaire

## Glossaire

**Bot** : Dans le cadre de jeux vidéo, les bots sont des adversaires artificiels qui miment le comportement de joueurs humains. Pilotés par l'ordinateur, ils permettent ainsi de jouer seul à un jeu multijoueur.

**FPS** : Un FPS (First Person Shooter, en Anglais) est un type de jeu vidéo de tir en 3D dans lequel le personnage doit en général éliminer des ennemis à l'aide d'une arme de tir et dans lequel l'angle de vue proposé simule le champ visuel du personnage incarné. On peut alors parler de jeux de tir à la première personne, mais également de shooter.

**IDE** : Integrated Development Environment. Programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication et souvent un débogueur.

**MOD** : Un mod (de l'anglais mod, abréviation de modification) est un jeu vidéo créé à partir d'un autre, ou une modification du jeu original, sous la forme d'une greffe qui se rajoute à celui-ci, le transformant parfois complètement.

**Unreal Tournament 2004** : Aussi connu sous l'abréviation UT2004, est un jeu vidéo de tir à la première personne futuriste, développé par Epic Games et Digital Extremes. Il s'agit d'un jeu de combats d'arènes, conçu principalement pour le multijoueur, bien que le jeu dispose d'un mode solo imitant des parties à plusieurs grâce à ses bots.

**UnrealEd** : UnrealEd (Unreal Editor, ou UEd en abrégé) est un logiciel permettant la création de carte de jeu (map en Anglais), pour les jeux de la série Unreal, et des jeux basés sur le moteur UnrealEngine. Ceci permet la création de nombreuses communautés de modélisme et la création d'une vaste quantité de cartes pour tous les jeux incluant UnrealEd. De plus, le jeu incorpore un langage de script, UnrealScript, qui permet aux modeleurs de paramétrer les niveaux au-delà du modélisme, ce qui permet donc d'accroître la longévité des jeux avec lesquels il est livré, étant donné que les communautés les complètent constamment avec de nouvelles cartes.

**UnrealScript** : L'UnrealScript est le langage de script utilisé par la série des Unreal, ainsi que d'autres jeux utilisant le même moteur. Il a pour but de :

- Supporter les principes de temps, d'état, de propriétés et de communications réseaux inexistantes nativement (mais abordables) dans des langages de programmation comme le C, le C++ et le Java, principales sources d'inspiration dans la création de ce langage.
- Fournir un langage simple, orienté objet et résistant. Ces caractéristiques ont largement contribué au succès des différents jeux dans la communauté de créateurs.
- Permettre une programmation de haut niveau, riche, avec des interactions entre objets plutôt que sur des bits et des pixels.

**WOTGreal** : WOTGreal est un IDE qui permet de développer des scripts en UnrealScript.

# Table des figures

# 1. Introduction

Au cours de la première année de Master Informatique à l'Université Montpellier 2, il est demandé aux étudiants d'effectuer un projet, pour une durée approximative de trois mois. Son but est de permettre aux étudiants d'effectuer un travail en groupe, avec les avantages et les contraintes que cela représente, tout en réalisant un projet concret correspondant à la spécialité choisie par les étudiants et en respectant les spécificités imposées par le sujet.

Ce projet, encadré par M. KORICHE, consiste à implémenter le comportement de Personnages Non Joueurs (PNJ ou bots) par des processus de décision dans un jeu de type shooter (équivalent à FPS, jeu de tir à la première personne). Nous devons donc développer l'intelligence artificielle (abrégée par le sigle IA) de ceux-ci, afin de les rendre capables de prendre des décisions dans leur environnement, tout en essayant de garder l'illusion que ces décisions sont prises par des personnages humains. Les PNJ doivent donc avoir un comportement stochastique, c'est-à-dire que le choix d'action qu'ils vont effectuer est déterminé selon une distribution de probabilités.

Dans ce TER, nous allons donc construire une maquette permettant de charger une carte avec son bâtiment 3D, ses objets et PNJ mais également construire l'intelligence artificielle des PNJ.

Dans la suite de ce rapport, nous allons vous expliquer les différentes étapes du développement de ce projet, avec tout d'abord l'analyse du sujet, puis l'architecture logicielle que nous avons choisie, suivi par notre choix pour la représentation de l'intelligence artificielle des bots, ainsi que leur intégration dans l'environnement de jeu, et pour finir, nous conclurons en présentant les problèmes que nous avons rencontrés.

## 1.1. Cahier des charges simplifié :

Au fur et à mesure du déroulement de ce TER, notre cahier des charges a subit quelques transformations, et ceci en accord avec M. Koriche. A la suite de ces transformations, nous avons fourni un travail équivalent à ce qui était demandé dans le sujet initial, mais avec une approche peut être moins traditionnelle.

Ci-dessous, vous trouverez donc notre cahier des charges tel que nous l'avions écrit, et au fur et à mesure de ce rapport, vous trouverez les modifications que nous y avons apportées.

- **Exigences fonctionnelles :**

Le projet devra regrouper toutes les fonctionnalités permettant, comme indiqué précédemment, de charger une carte complète en 3D (terrain, objets et PNJ) et surtout de construire les intelligences artificielles des personnages non joueurs.

- **Charger une carte (terrain ou map) :**

Une carte devra être constituée d'un environnement simple dans un premier temps : un terrain plat sans étage avec peu d'obstacles, tout cela en 3D. L'intégration des PNJ se fera par la suite.

- **Construire l'intelligence :**

Les PNJ devront avoir un comportement stochastique : nous souhaitons que les personnages se déplacent et réagissent de manière inattendue pour un joueur humain. Nous devions initialement implémenter les comportements des PNJ à l'aide de machines markoviennes (comportement probabilistique) qui pourront être modifiées (variations des probabilités) en fonction d'événements. Nous verrons plus tard dans ce rapport que nous avons choisi de mettre en place une autre solution, un planificateur.

- **Exigences de qualité :**

Nous avons choisi de ne pas investir beaucoup de temps dans le moteur de jeu (regroupant les moteurs graphique et physique entre autres) pour se consacrer plus intensément à la partie de l'IA des PNJ.

En d'autres termes, nous avons choisi d'utiliser un moteur de jeu déjà existant, en partie car la programmation d'un tel module représente beaucoup de travail (contrainte de temps). Le fait d'en créer un nous aurait contraint à passer moins de temps sur la conception de l'IA des PNJ, ce qui nous intéresse principalement. Ce choix est, de plus, guidé en grande partie par nos connaissances actuelles dans les domaines de la 3D et de l'IA.

Nous ne nous focaliserons pas non plus sur l'aspect design du jeu. Cela ne veut pas dire pour autant une absence de recherche dans l'ambiance du jeu ou encore la gestion des graphismes (nous développerons d'avantage cette partie si nous arrivons à un résultat satisfaisant concernant l'IA des bots rapidement).

Le shooter devra néanmoins rester attirant et simple, permettant aux spectateurs de profiter pleinement du spectacle (le combat des bots).

- **Exigences de performance :**

Le programme devra être fluide, c'est à dire que les personnages devront se déplacer sans à-coups. Nous devrons pouvoir mettre dans le jeu deux PNJ (ou plus dans l'idéal) ayant un comportement distinct (nous créerons différentes classes de personnages), sans diminuer la performance du programme.

- **Contraintes de conception :**

Nous essaierons de faire en sorte que notre projet fonctionne sur un maximum de machines (au niveau des performances et non de l'OS). Nous ferons aussi en sorte que le temps de chargement reste raisonnable.

## 1.2. Analyse du projet :

- **Choix du moteur de jeu :**

Avant de commencer l'analyse du sujet, il nous a fallu choisir le moteur de jeu que nous allions utiliser. En effet, comme nous l'avons dit, nous souhaitions partir d'une base solide, afin de pouvoir respecter la contrainte de temps imposée, et non pas développer en plus de l'IA des bots, notre propre moteur de jeu.

Pour cela, nous nous étions arrêtés sur deux moteurs de jeu possibles :

- l'Id Tech III (aussi connu sous le nom de Quake 3 Engine), développé par Id Software et à l'origine de nombreux jeux.

- l'UnrealEngine 2 (ou Warfare Engine), à l'origine de Unreal Tournament 2003, mais également de nombreux autres jeux.

Ces deux moteurs de jeu (regroupant moteurs graphique et physique) ont été largement utilisés et sont aujourd'hui tombés dans le domaine libre.

Finalement, et après en avoir discuté avec M. Koriche, nous avons choisi d'utiliser l'UnrealEngine 2, beaucoup plus utilisé à l'heure actuelle dans le domaine professionnel, et disposant de nombreux outils de développement.

- **Premiers pas :**

Afin d'avoir une idée générale du sujet qui nous a été confié, nous avons commencé par réaliser une analyse comprenant un diagramme des cas d'utilisation, pour nous offrir un aperçu des fonctionnalités à développer pour nos bots, puis un diagramme des classes, pour nous donner une idée de la structure de l'application.

Ci-dessous, vous trouverez le diagramme des cas d'utilisation, qui permet d'illustrer les principales fonctions attendues pour un bot.

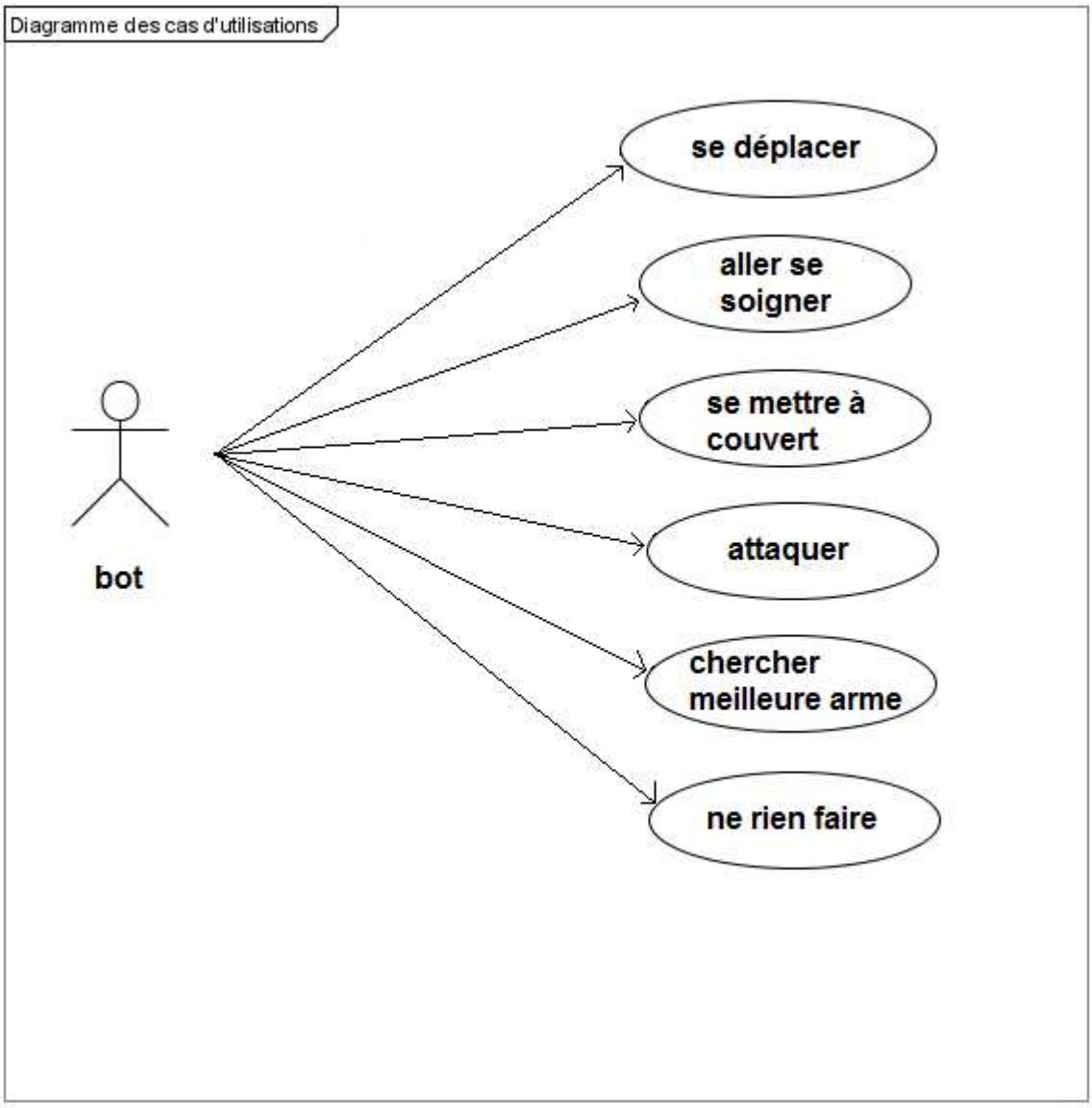



*Figure 1 : Diagramme des cas d'utilisation.*

En regardant le diagramme des cas d'utilisation, vous pouvez voir que nous avons voulu rendre nos bots les plus humains possible, en leur offrant l'opportunité de réaliser la majorité des actions qu'utilise un joueur humain, afin de ne pas déséquilibrer le jeu.

Bien entendu, il est possible d'ajouter encore de nombreuses actions, pour renforcer l'immersion dans le jeu, mais celles-ci constituent déjà une base essentielle.

Par la suite, nous avons développé notre diagramme de classes au fur et à mesure de l'avancement du projet, puisque nous sommes partis d'une base déjà existante : le moteur d'Unreal. Pour cela, il nous a fallu comprendre le fonctionnement du programme ainsi que la hiérarchie des classes, afin d'y implémenter les nôtres.

A partir de cette première analyse, nous avons décidé de rendre notre code modulable. En effet, lorsque nous avions commencé à modifier les classes existantes, nous nous sommes rendu compte que nous étions assez vite bridés dans nos envies, et qu'il était difficile de transformer le code existant. Nous avons donc pris la décision de développer du code qui pourrait être réutilisé. Ceci sera expliqué plus tard dans ce rapport.

Pour une question de lisibilité, nous avons placé le diagramme des classes en annexe de ce rapport (page 41).

## 1.3. Organisation du travail :

A la suite de cette analyse, nous en sommes arrivés à découper le projet en plusieurs tâches, visibles dans le diagramme de Gantt ci-dessous :

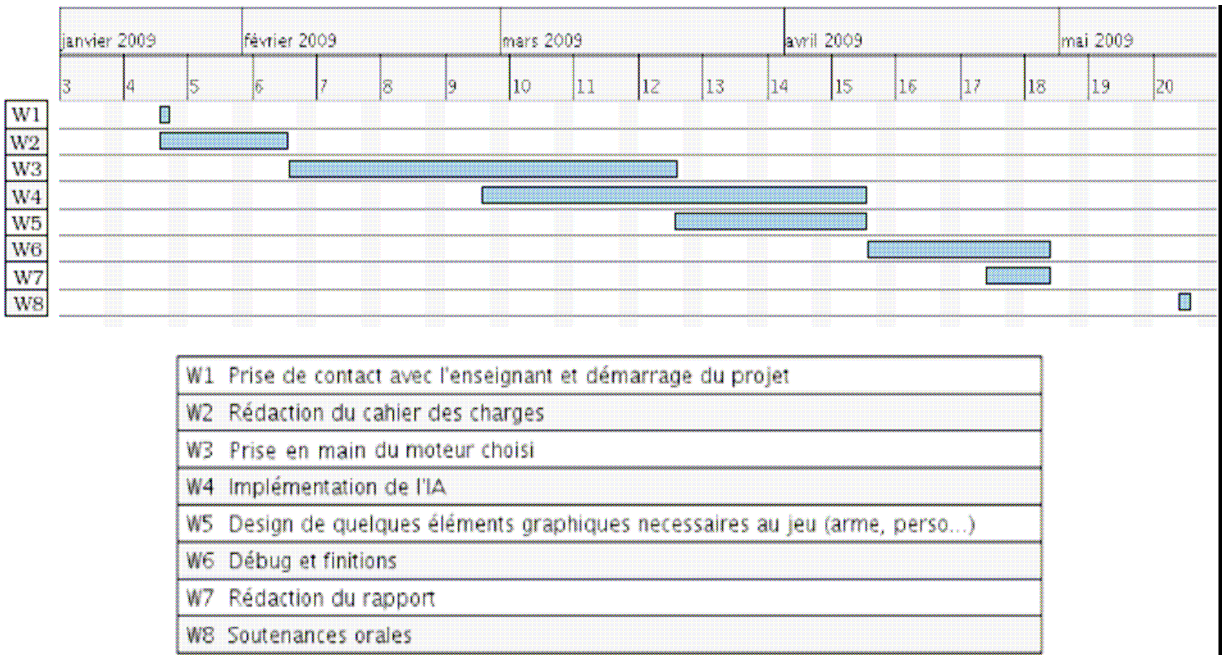



*Figure 2 : Diagramme de Gantt.*

Au cours de ce projet, nous nous sommes réunis de nombreuses fois, afin de discuter de l'avancement de chacun, de nous assurer de la compatibilité entre chaque partie et de nous entraider.

Nous avions également mis en place un forum et un FTP sur Internet, afin de partager facilement notre travail avec le reste du groupe, mais également poser des questions, donner son avis, … Mais nous avions aussi mis en place cet outil pour ne pas répéter une erreur que nous avions commise au début de ce projet. En effet, suite à une panne informatique, nous avions perdu la version finale de notre cahier des charges et nous n'avons donc pas voulu connaître le même problème une deuxième fois.

# 2. Architecture logicielle

## 2.1. Explication des classes :

La totalité des classes pour ce projet a été écrite en UnrealScript. Les composantes principales du projet sont listées ci-dessous.

- **TERAIController :**

Cette classe est une extension de *ScriptedController* et est responsable de l'intelligence du bot. Elle permet notamment de faire le lien entre toutes les classes permettant au bot de réfléchir (planifier) et agir.

Dépendances : Action, Goal, WorldState, Sensor, PathNode, Planner.

- **TERPawn :**

Le Pawn, dans Unreal, est une classe permettant d'avoir une représentation physique du bot lors de la création d'une map. C'est l'interface entre la future intelligence et le créateur : il peut en effet choisir quelles actions et buts composeront le bot.

Dépendances : TERAIController, Action, Goal.

- **TERGameInfo :**

Cette classe permet simplement de renseigner le jeu Unreal Tournament 2004. Nous lui disons d'utiliser nos classes de bot (Pawn et Controller) ainsi que nos maps. Elle est une sous-classe de *DeathMatch*, permettant de partir d'un mod de jeu très semblable à celui que l'on veut construire.

Dépendances : Aucune.

- **AISymbolBase :**

Nous avons choisi de nous servir d'une classe abstraite de base, pour les Actions et Goals, car ces classes devront utiliser toutes deux les structures décrites dans celle-ci (notamment les états). Cette classe n'est donc pas indispensable mais permet néanmoins d'apporter une certaine clarté au diagramme de classes.

Dépendances : Aucune.

- **WorldState :**

Définie en partie dans la classe précédente (AISymbolBase), cette classe permet de clarifier ce qu'est vraiment l'état actuel de l'environnement, mais aussi de faire partie intégrante de certaines classes étant elles-mêmes sous-classes d'AISymbolBase. Nous définissons ici aussi, plus précisément, la structure basée sur une structure de paires : soit Actor et valeur, soit propriété et booléen (nous pourrons, si l'on en a besoin, ajouter de nouveaux types pour déclarer plus précisément certains éléments de l'environnement).

Dépendances : Aucune.

- **Goal :**

La classe Goal est la classe abstraite de base pour la totalité des buts que l'on pourra imaginer et écrire, et que les agents essaieront d'atteindre. Une classe représentant un but à atteindre sera juste définie par les propriétés à atteindre (environnement souhaité). Un bot choisira le but à atteindre grâce à une fonction qui sera redéfinie par chaque classe de Goal : EvaluateGoalRelevancy (qui calculera la pertinence de ce but par rapport à l'environnement courant).

Dépendances : WorldState, TERAIController.

- **Action :**

La classe Action est la classe abstraite de base pour la totalité des actions qu'un bot peut réaliser (courir, tirer sur un autre bot, se cacher, …). Ces actions seront utilisées par le planificateur pour essayer d'atteindre un but. Chaque action sera définie par des préconditions (qui devront être vraies pour que l'action puisse être réalisée) et des effets (qui prendront effets dès que l'action aura été correctement réalisée). C'est aussi dans chaque classe définissant une action, que l'on précisera l'exécution de celle-ci (appels aux méthodes de déplacement, tir, … déjà définies dans Unreal).

Dépendances : WorldState, TERAIController.

- **Planner :**

Cette classe implémente le planificateur qui se sert d'un algorithme A* pour trouver un plan cohérent (pour arriver au but voulu), prenant en compte l'ensemble des actions que le bot peut réaliser. Cette classe se veut la plus générique possible pour pouvoir être adaptée à différents types d'IA, d'actions et de buts (on pourra ainsi programmer d'autres types de jeux se servant du même planificateur : IA hiérarchique, collective, …).

Dépendances : Action, Goal.

## 2.2. Pathfinding :

### 2.2.1. Principe :

- **L'existant :**

Dans le jeu Unreal Tournament 2004, les bots n'ont pas la capacité d'interpréter l'environnement 3D qui les entoure. Aussi, pour naviguer dans cet environnement, les développeurs ont intégré un système de nœuds (nodes en Anglais) qui, reliés entre eux, forment un réseau que le bot peut emprunter.

Ces réseaux sont en général faits manuellement par les programmeurs, car la génération automatique des réseaux donne de médiocres résultats. Il existe plusieurs « couleurs » de réseaux, représentant les possibilités offertes aux bots, tels que monter à une échelle, emprunter un téléporteur, … Mais ces couleurs sont également censées indiquer si un chemin est pertinent à suivre ou non.

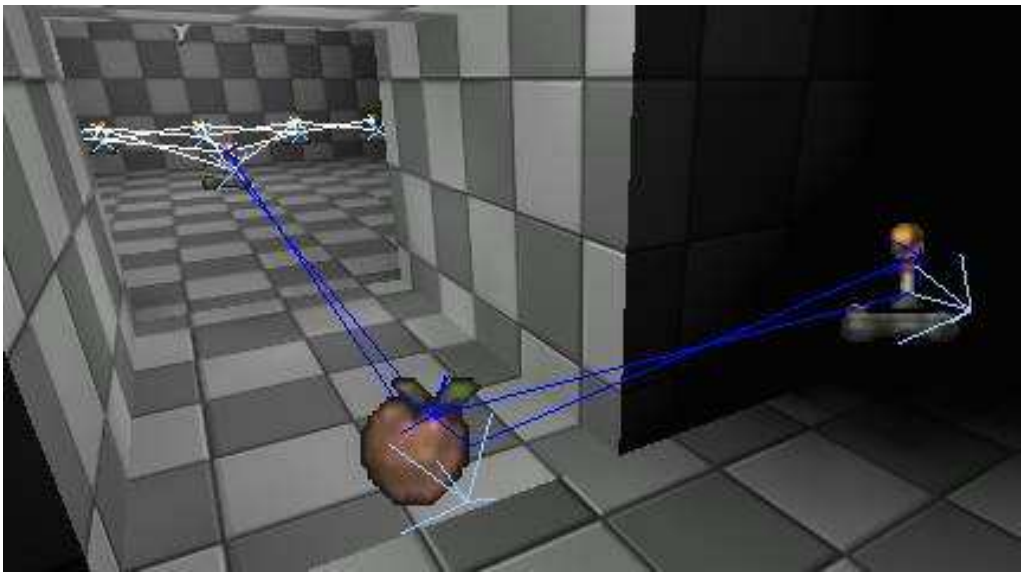


*Figure 3 : Exemple de nœuds.*

- **Ce que nous avons fait :**

A partir du principe de nœuds existant, nous voulions utiliser l'algorithme A* (ou A Star en Anglais) pour déterminer les nœuds à emprunter en fonction de la situation. Cependant, ce système de nœud est incompatible avec l'implémentation que nous proposions. Nous avons donc, pour remédier à cela, implémenté des nouveaux types de nœuds, à insérer à la place des « PathNode » initialement prévus : les « AStarPathNode ».

Nous avons alors créé une sous-classe de « PathNode » pour pouvoir directement placer ces nœuds sur la carte, dans l'éditeur (et aussi pour éviter les erreurs de compilation de maps), mais également pour rester dans l'esprit de la construction d'une extension du jeu. En faisant cela, nous avons pu rajouter aux variables déjà présentes un coût afin de définir le danger que représente le nœud si l'on passe par celui-ci, dans le cas d'un endroit très exposé par exemple, mais aussi des attributs utiles aux calculs des divers chemins possibles (valeurs de G et H en ce nœud, durant le calcul).

### 2.2.2. Algorithme utilisé :

- **Présentation :**

L'algorithme de recherche de chemin A* utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin passant par celui-ci. Cet algorithme itératif est simple car il ne nécessite pas de prétraitement et pas (ou peu) de mémoire.

- **Description :**

A* commence au nœud initial : la position du bot. L'algorithme calcule ensuite la distance qui sépare ce nœud du but à atteindre. La somme du coût (G lors de la première itération, est nul) et de l'évaluation (H) représente le coût heuristique assigné au chemin menant à ce nœud. Le nœud est alors ajouté à une file d'attente prioritaire, couramment appelée *open list*.

L'algorithme retire le premier nœud de l'*open list*. Si la file liste est vide, il n'y a aucun chemin du nœud initial au nœud d'arrivée, ce qui interrompt l'algorithme : il n'existe aucun chemin entre notre point de départ et celui d'arrivée. Par contre, si le nœud retenu est le nœud d'arrivée, A* reconstruit le chemin complet et l'algorithme s'arrête : il a trouvé un chemin cohérent. Pour cette reconstruction on se sert d'une partie des informations sauvées dans la *closed list*.

Si le nœud n'est pas le nœud d'arrivée, de nouveaux nœuds sont créés pour tous les nœuds voisins admissibles (on ne regarde, dans notre cas, que les nœuds qui sont directement visibles par le nœud courant). Pour chaque nœuds successifs, A* calcule son coût et le stocke avec le nœud. Ce coût est calculé à partir de la somme du coût de son ancêtre et du coût de l'opération pour atteindre ce nouveau nœud. De plus, on stocke la valeur de l'évaluation H (on a choisi la distance à vol d'oiseau entre le nœud courant et le but) ainsi que le nœud parent (celui par lequel on est arrivé au nœud courant).

L'algorithme maintient également la liste de nœuds qui ont été vérifiés dans la *closed list*. Si un nœud nouvellement produit est déjà dans cette liste avec un coût égal ou inférieur, aucune opération n'est faite sur ce nœud ni sur son homologue se trouvant dans la liste.

Ensuite, l'évaluation de la distance du nouveau nœud au nœud d'arrivée est ajoutée au coût pour former l'heuristique du nœud. Ce nœud est alors ajouté à la liste d'attente prioritaire, à moins qu'un nœud identique dans cette liste ne possède déjà une heuristique inférieure ou égale.

Une fois les trois étapes ci-dessus réalisées pour chaque nouveau nœud contigu, le nœud original pris de la file d'attente prioritaire est ajouté à la liste des nœuds vérifiés. Le prochain nœud est alors retiré de la file d'attente prioritaire et le processus recommence.

Concrètement, une fois le nœud final atteint, il suffit de regarder dans la *closed list* les nœuds ayant permis de trouver ce chemin en regardant à chaque fois les parents, stockés directement dans le nœud, pour arriver jusqu'au nœud initial.

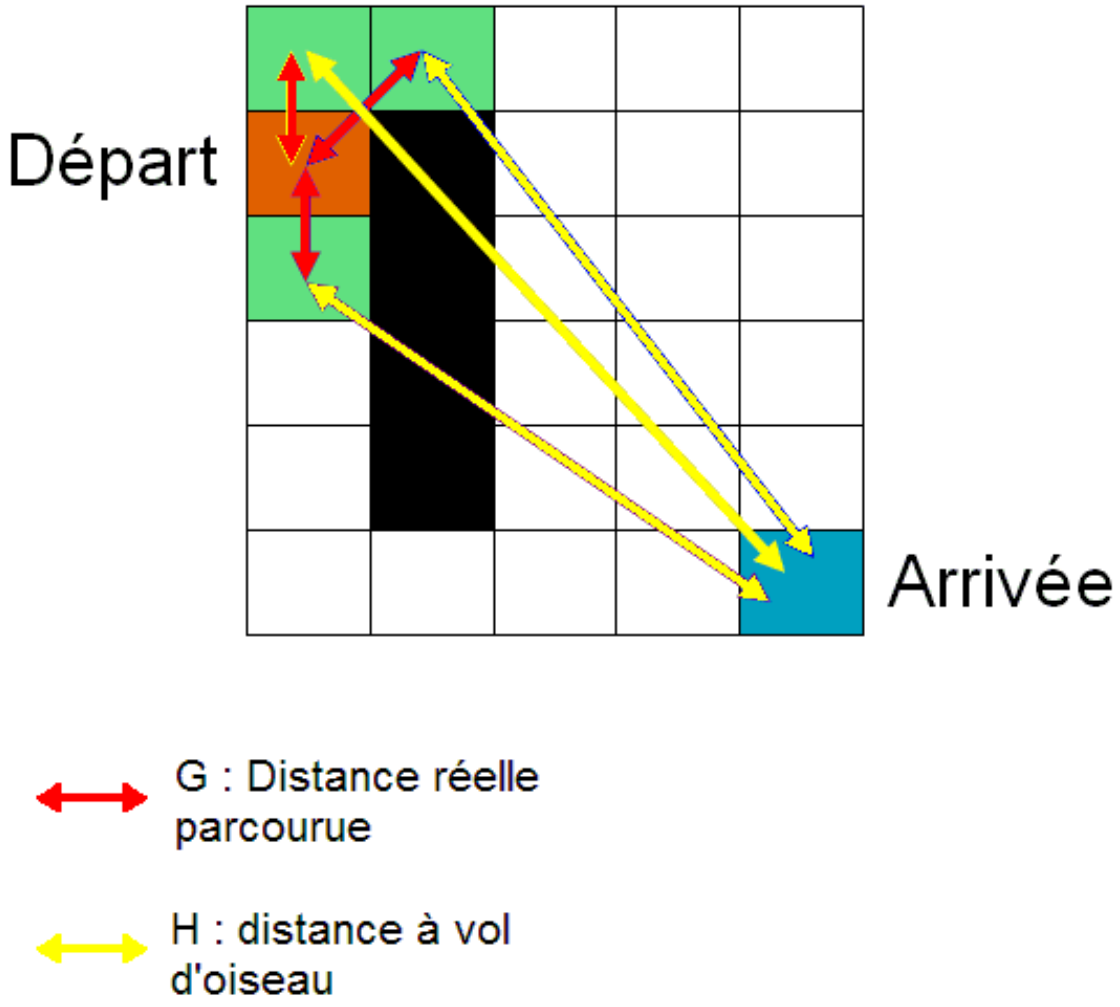



*Figure 4 : Principe de A*.*

- **L'algorithme A* :**

Voici donc l'algorithme que nous avons utilisé :

Sommet source (S)
Sommet destination (D)
Liste des sommets à explorer (E) : sommet source S
Liste des sommets visités (V) : vide

Tant que (la liste E est non vide) et (D n'est pas dans E) Faire
    Récupérer le sommet X de coût total F minimum
    Ajouter X à la liste V
    Ajouter les successeurs de X (non déjà visités) à la liste E en évaluant leur coût total F et en identifiant leur prédécesseur.

    Si (un successeur est déjà présent dans E) et (nouveau coût est inférieur à l'ancien) Alors
        Changer son coût total
        Changer son prédécesseur
    Fin Si
Fin Tant que

# 3. Planificateur

Comme nous l'avons expliqué dans le cahier des charges de ce rapport, nous avons fait évoluer le sujet de base, afin de présenter quelque chose de moins traditionnel.

- **Prise de décision traditionnelle :**

En règle générale, dans les jeux vidéo, la prise de décision est un processus statique, utilisant l'architecture des machines à états finis (Finite State Machine, ou FSM), dans laquelle le comportement des PNJ est entièrement déterminé par son état, et les règles d'évolution entre les états sont prédéfinies. Or, étant donné la nature très déterministe de ce type d'architecture, il est facile pour un joueur humain d'apprendre ou de prévoir ce que vont faire les PNJ, et développer ainsi des stratégies qui exploitent cette faille.

C'est à partir de ce constat que nous avons choisi de faire évoluer cela, en implémentant un planificateur.

- **Planificateur :**

Le planificateur est un domaine relativement nouveau dans le domaine du jeu vidéo, puisqu'il utilise une approche fondée sur les objectifs. Il faut donc que son utilisation se fasse en un temps raisonnable, avec des performances au moins équivalentes aux techniques d'intelligence artificielle classiques.

L'objectif d'un PNJ va être d'essayer d'atteindre un but. Dans notre cas, son objectif principal sera de tuer les adversaires (cas du deathmatch), mais il peut également avoir besoin de se soigner ou de se mettre à couvert. Afin d'atteindre son but, notre PNJ va utiliser une séquence d'actions, appelée plan. C'est cet enchaînement qui va lui permettre d'atteindre son objectif.

Avant de pouvoir calculer un plan, le PNJ doit d'abord sélectionner un but qu'il va tenter d'atteindre. Chacun des buts est donc associé à une fonction qui retourne une valeur comprise entre 0 et 1, en fonction du contexte et de l'importance qu'accorde l'utilisateur à chacun des objectifs. Par exemple, pour satisfaire l'objectif « tuer un ennemi », le bot pourra enchainer plusieurs actions telles que « courir vers un nœud » afin de se déplacer, puis « faire face à sa cible » une fois qu'il l'aura aperçu et enfin « tirer ».

En général, le planificateur inclut un état courant de l'environnement, un but présélectionné et une suite d'actions. Le plan est formulé en recherchant un chemin dans le graphe d'états, composé de la plus petite suite d'actions qui permettent d'atteindre le but à partir de l'état courant. Cette liste des états du planificateur est implémentée en utilisant un tableau dynamique redimensionnable. Elle est codée en UnrealScript et est l'équivalent d'un vector en C++.

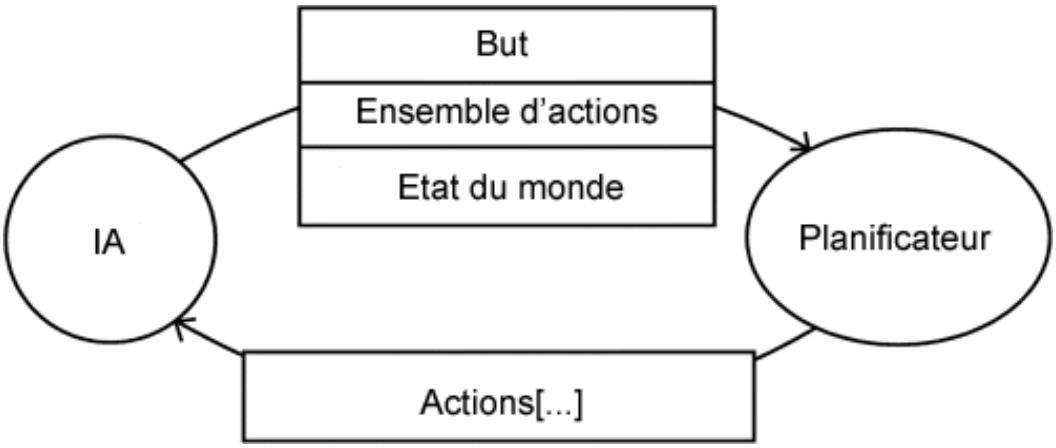



*Figure 5 : Schéma du planificateur.*

L'utilisation d'un planificateur comporte des avantages intéressants au niveau du gameplay et du développement, comparé à une IA classique, comme l'utilisation de machines à état finis, le planificateur permet de voir des réactions intelligentes venant des PNJ vis-à-vis de scénarios inattendus.

Au niveau du développement, l'un des principaux avantages se situe au niveau de l'implémentation des actions qui se révèle particulièrement simple et pouvant être réutilisées de manière modulaire. De plus, planifier une séquence d'actions peut se révéler plus coûteux en terme de calcul mais permet de représenter des attitudes plus complexes car le processus de planification mime la manière dont les humains résolvent les problèmes.

A partir de ce constat, nous voyons bien qu'il est nécessaire de mesurer les performances de cette solution lorsqu'on utilise un grand nombre d'agents. En effet, il ne faut pas que le joueur puisse percevoir de latence, le temps que le bot calcul son nouveau plan, ou que plusieurs bots souhaitent calculer leur plan (moment pendant lequel le PNJ reste bloqué, sans effectuer d'actions).

La solution à ce problème consiste à faire en sorte que les plans soient interruptibles (principe de l'algorithme anytime), puis que la recherche d'un nouveau plan soit lancée, tout en conservant les données déjà calculées précédemment. Ainsi, la planification est centrée sur l'état du personnage, qui est mis à jour par ses différents senseurs. Chacun d'entre eux a la possibilité d'interrompre le plan, si les modifications perçues ont une importance. Dans ce cas, l'objectif est recalculé et le PNJ va utiliser ce nouveau plan.

Puis, nous avons également mis en place une file afin de gérer le cas où plusieurs agents demandent le calcul d'un plan simultanément : lorsqu'un agent doit calculer un nouveau plan, il fait une demande de planification auprès de l'ordonnanceur qui va se charger de le placer dans une file d'attente FIFO. De plus, le nombre d'itérations possibles lors du calcul du plan est prédéfini. Ceci permet donc de ne pas calculer des plans trop longs (en général, ils n'auront de toute façon pas le temps de s'exécuter dans leur intégralité) et empêche également les temps de latence.

Quand le bot est dans la file pour une demande de calcul de plan, il reste inactif. Si cela pose problème ou si cet état devient trop long, une solution serait de fournir un plan statique composé d'une seule action avec un but commun à tous les PNJ, que le bot pourrait déclencher au cours de l'attente. Pour éviter les phases inactives des bots, il serait également intéressant de planifier des sous- buts et de les planifier récursivement jusqu'à ce que chaque but soit résolu comme des actions atomiques.

Une autre optimisation consisterait à mettre en place un système de priorités où un PNJ plus proche d'un objectif général serait prioritaire pour le calcul d'un plan.

- **Planification par A* :**

Chercher un chemin dans un plan possède de nombreuses similitudes avec le pathfinding dans les jeux. Ainsi, l'algorithme A* peut être utilisé par le planificateur avec :

  - les états qui représentent les nœuds,
  - les actions qui représentent le passage entre deux états.

Le coût d'un nœud est donc la somme des coûts des actions pour atteindre cet état. Le coût de l'heuristique depuis un nœud peut être calculé comme la somme des propriétés de l'état but insatisfait pour ce nœud.

Le comportement d'un PNJ peut donc être modifié en attribuant de préférences des actions de coût minimum.

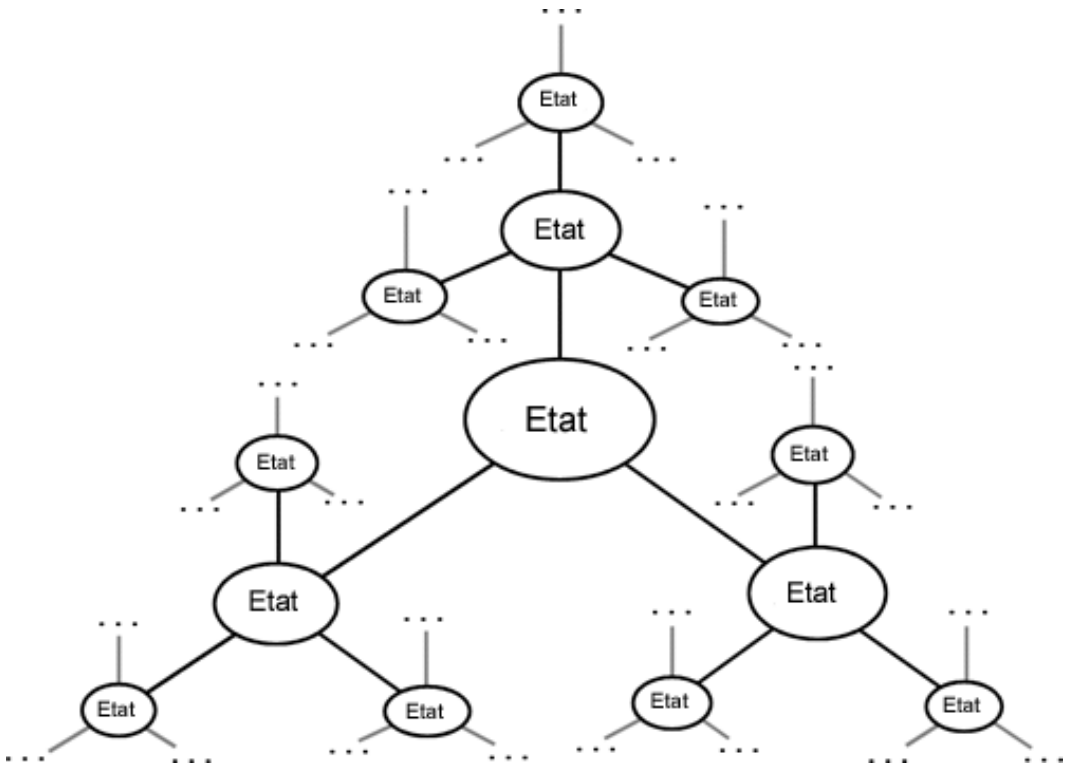



*Figure 6 : Représentation d'un graphe d'état.*

Pour réduire la taille (potentiellement illimitée) du graphe parcouru, il est préférable de partir du but à atteindre pour arriver à l'état courant, en ajoutant des actions qui satisfassent des parties du but à atteindre ou des pré conditions aux actions déjà ajoutées.

# 4. Intégration du code

## 4.1. Création de cartes et de mods :

Nous avons choisi d'utiliser l'UnrealEd, fourni avec l'UnrealEngine 2, afin de nous assurer de la pleine compatibilité avec le moteur. Cet éditeur est l'un des plus utilisés dans le domaine du jeu vidéo et dispose de nombreux tutoriels en ligne, ainsi que d'une communauté forte et productive.

Nous avons alors commencé par développer de petites cartes de jeu simples et sans prétention, pour mieux nous familiariser avec les objets du jeu, tels que les armes, les points d'apparition des bots, …

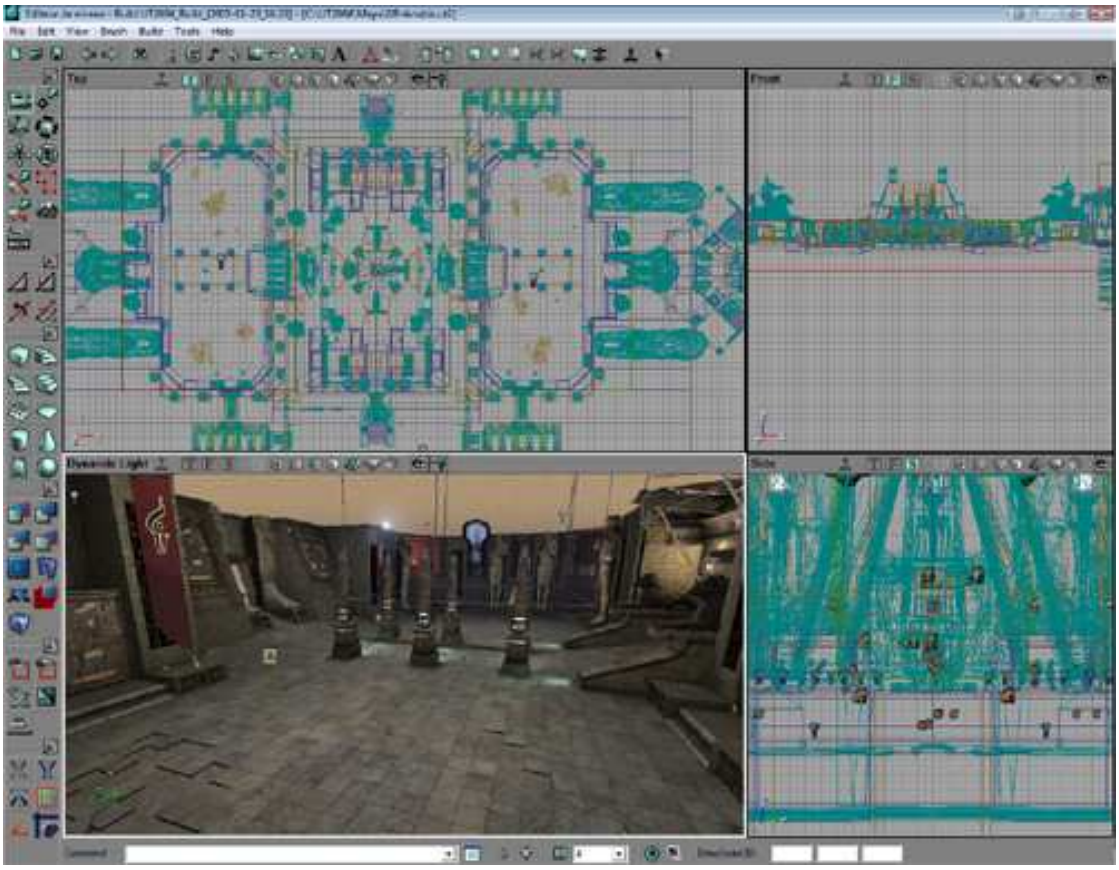


*Figure 7 : Capture écran de l'UnrealEd*

Comme vous pouvez le voir, UnrealEd, comme la plupart des éditeurs de jeu, est divisé en 4 fenêtres principales. Trois d'entre elles présentent une vue de la carte sous forme de plan, et une quatrième permet de visualiser directement le résultat dans une vue 3D.


*Figure 8 : Résultat dans le jeu Unreal*

Dans cette capture écran, vous pouvez voir le résultat dans le jeu Unreal de la carte développée dans UnrealEd.

## 4.2. Intégration des PNJ dans le jeu :

Pour mettre en place des PNJ sur les cartes de jeux, développées par nos soins ou non, l'utilisateur doit les intégrer directement dans la map à l'aide l'UnrealEd. Là où ce choix est intéressant, c'est que pour chacun d'entre eux, le joueur peut fixer les actions et les objectifs qu'il souhaite.

Grâce à cela, des comportements bien distinct peuvent être créés pour nos PNJ, avec par exemple un bot chargé de l'attaque, un autre plutôt défensif, ou encore un bot qui soignerait les autres, un bot sniper, …

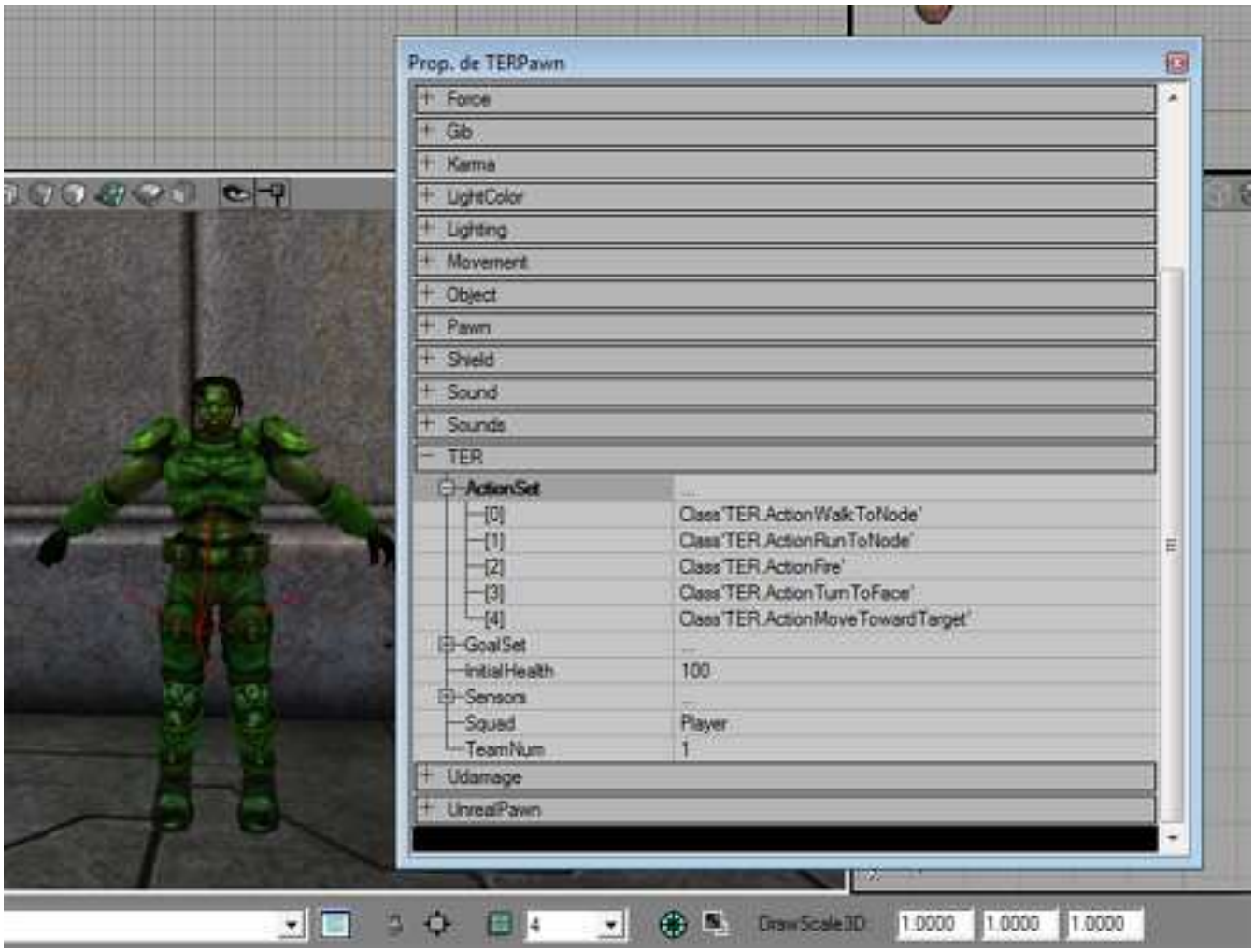



*Figure 9 : Attribuer des actions dans UnrealEd*

Dans cet exemple, nous avons donné plusieurs actions à notre PNJ, dont notamment la possibilité de marcher vers un nœud, de courir, de tirer, ...

# 5. Conclusion

## 5.1. Phase de tests :

Pour bien comprendre comment fonctionnait la planification, il nous a semblé utile de calculer la complexité d'une telle tâche, afin de pouvoir choisir l'algorithme à implémenter. Cela relève autant du calcul que du test, c'est pourquoi nous avons choisi de mettre ces réflexions dans cette partie.

Tout d'abord, rappelons que le fait de planifier est de partir d'un état initial pour arriver dans un état final. La modélisation la plus courante est un graphe, où chaque nœud représente un état de l'environnement et chaque arête une action qui permet de passer d'un état du monde à un autre.

La complexité de l'élaboration d'un graphe d'états dépend de :

- **b** : le facteur de branchement. Autrement dit le nombre moyen de nouveaux états qui peut être engendré à partir d'un état.
- **d** : la profondeur minimum entre l'état initial et l'état final.

On peut voir que le facteur de branchement **b** dépendra avant tout du nombre d'éléments qui serviront à décrire l'environnement, implicitement lié au nombre d'actions qu'un bot peut réaliser. En effet, nous essaierons, la plupart du temps, de modéliser les actions les plus simples, tant par facilité de compréhension, de codage que dans l'intérêt de la planification (créer des actions complexes aux effets multiples ne ferait qu'alourdir une élaboration de plan, certaines actions pourraient avoir des effets contraires et ne feraient qu'allonger le schéma de conception). La profondeur **d**, quand à elle**,** dépendra du nombre d'itérations que nous permettrons à notre planificateur de réaliser.

À partir de cette petite réflexion, nous avons pu choisir l'algorithme à implémenter pour réaliser notre planificateur.

Avec un parcours par niveau, la solution serait trouvée en $O(b^d)$. L'algorithme de parcours en profondeur nous donnerait une complexité semblable mais il faudrait donner dans ce cas une profondeur maximum de « retour arrière ». La solution la plus appropriée et la plus simple à mettre en place est donc autre : il faut se tourner à nouveau vers A*. En effet, la recherche d'un plan ne diffère pas beaucoup de celle d'un chemin si le problème est correctement modélisé. Il nous suffit donc de choisir correctement l'heuristique pour que cela réduise le facteur de branchement.

Ainsi, nous avons une complexité en $O(c^d)$, où c représente le facteur de branchement réduit (dépendant toujours du nombre d'actions et à la précision de la description de l'environnement dans lequel évolue les bots).

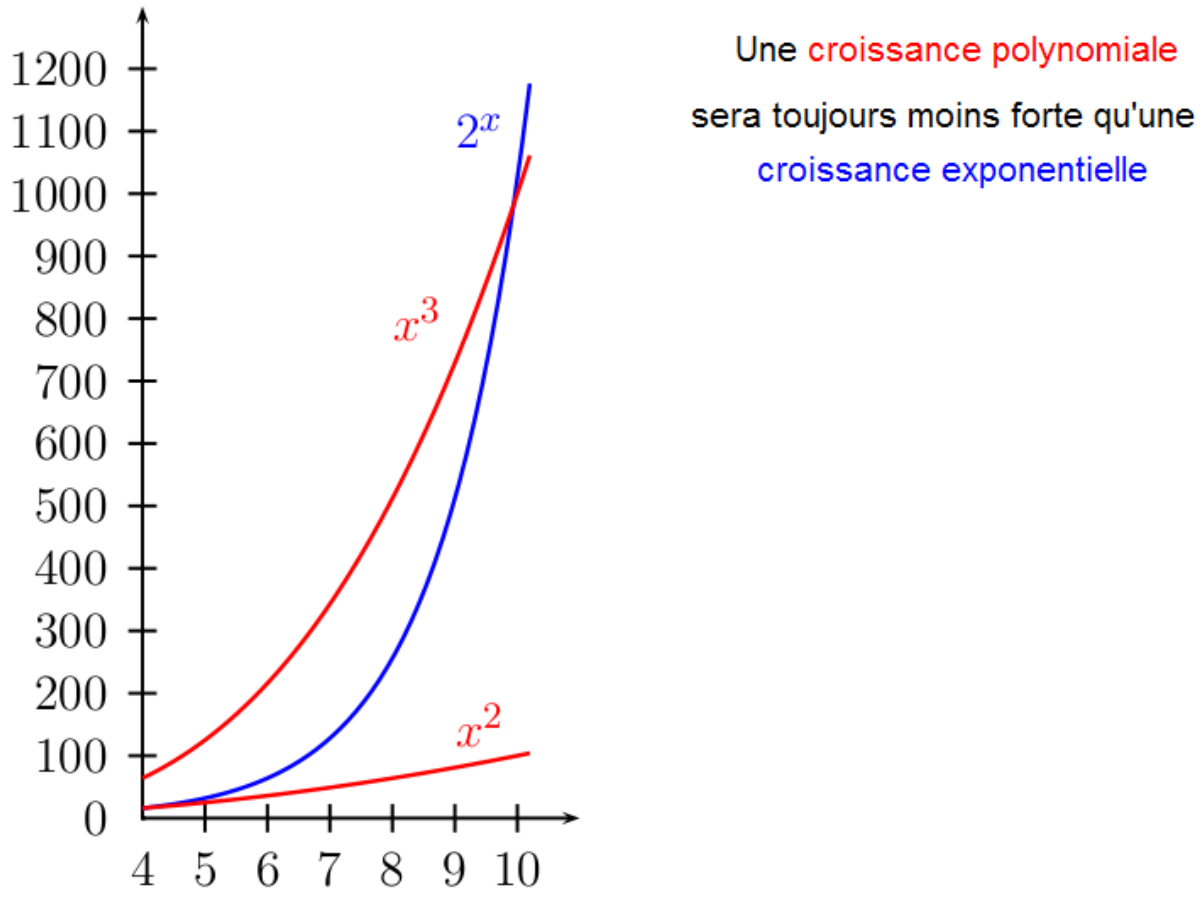



*Figure 10 : Comparaison des croissances*

Nous pouvons donc voir que nous obtenons une complexité polynomiale pour l'heuristique permettant de rechercher dans un espace d'états plus grand.

Pour nous assurer de l'efficacité de notre programme, et en particulier du planificateur, nous avons lancé celui-ci de nombreuses fois, en faisant varier plusieurs paramètres.

Nous avons donc commencé par lancer un nombre de bots de plus en plus grand, afin de nous assurer qu'aucun d'entre eux ne restait bloqué sur place, le temps que le planificateur calcule un plan pour chacun. Il s'est avéré que cela n'a posé aucune difficulté.

Ensuite, nous avons augmenté le nombre d'actions réalisables par nos bots, pour s'assurer que l'algorithme A*, malgré une taille du graphe d'état du planificateur de plus en plus importante, arriverait à trouver une solution. Là encore, ceci n'a pas posé de problèmes, mais pas à cause de la puissance du planificateur, mais plutôt à cause de la proximité des actions et des buts : bien souvent, il suffit de très peu d'actions pour réaliser un but (une ou deux tout au plus).

Nous avons également modifié le nombre d'itérations de l'algorithme A* pour trouver une solution dans le graphe d'état, et ainsi, confronter deux types de bots, l'un avec un plan complet, et les autres avec un plan très court. Malheureusement, le nombre d'actions réalisables par les bots, bien qu'important, reste très proche du nombre de buts à atteindre dans ce type de jeu (FPS). C'est pourquoi les plans des deux bots sont très semblables (voire identiques) dans la plupart des cas. Il serait plus judicieux de faire un test semblable dans un type de jeu où il y aurait beaucoup plus d'actions que de buts (comme par exemple dans les RPG). Cela nous permettrait de noter des écarts plus importants dans la ligne de conduite des bots de RPG, en comparaison aux bots de FPS (où le ratio Action/Goal est très proche de 1 en règle générale).

Un autre moyen efficace pour réaliser ce test sur la planification d'actions serait d'agrandir considérablement le nombre d'actions possibles (ajouter des objets à déplacer) et de modéliser encore plus fidèlement l'environnement courant. De ce fait, pour tester notre planificateur, nous avons juste lancé un pathfinding (l'algorithme et la planification se faisant sur un modèle semblable). Dans ces cas, les résultats sont identiques aux prévisions : un bot pouvant voir plus loin trouvera un chemin meilleur que le bot se limitant à peu d'itérations, même dans le pire graphe pour A*.

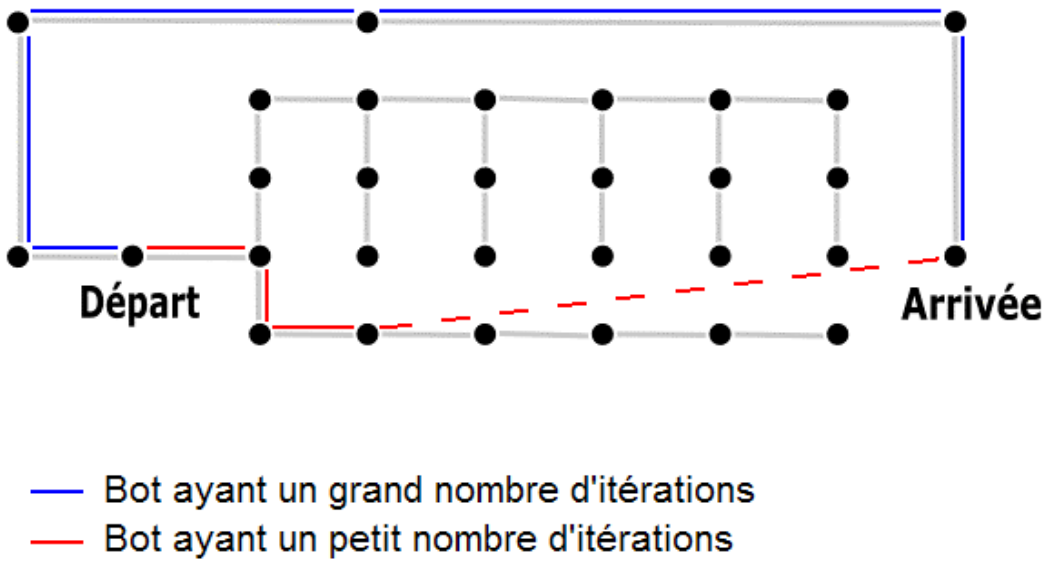



*Figure 11 : Comparaison de A* en fonction du nombre d'itérations*

Et pour terminer cette phase de tests, nous nous sommes également confrontés à nos bots, pour vérifier leur potentiel, et bien qu'encore en phase de développement, ils présentent déjà un certain intérêt de jeu.

En conclusion, nous sommes plutôt satisfait de notre travail sur le planificateur, puisque celui-ci se révèle tout à fait efficace et conforme à nos attentes, rendant ainsi nos bots un peu plus imprévisibles que la moyenne.

## 5.2. Problèmes rencontrés :

Le premier problème que nous avons rencontré concerne l'implémentation de nos bots. En effet, lors de nos essais, nous placions un de nos bots sur la carte de jeu, et nous testions son comportement. Mais nous obtenions alors deux bots, l'un réagissant comme prévu, et l'autre se comportant comme un bot développé par les créateurs du jeu. Ce problème venait en fait de nos bots qui héritaient d'une mauvaise classe mère. Toutefois, ce problème a été résolu lorsque nous avons réécris nos propres bots.

Le problème suivant a été celui de l'utilisation des nœuds pour l'implémentation du pathfinding. Au début, nous pensions utiliser le langage à script UnrealScript pour nous permettre de donner nos ordres aux bots, mais nous n'avons pas trouvé suffisamment de documentation à ce sujet pour nous permettre de résoudre ce problème. Toutefois, ce problème de nœuds fut résolu en codant nos propres nœuds que nous utilisons pour le pathfinding de nos bots.

Enfin, nous avons rencontré un dernier problème, pour le moment encore non encore résolu, et qui fait que lorsqu'un bot meurt, il ne réapparait pas sur la carte de jeu. En effet, en réécrivant nos bots, nous sommes remontés à un niveau élevé dans la hiérarchie des classes, et nous ne pouvons plus faire appel à cette propriété.

## 5.3. Discussion :

Au cours de ce projet, nous nous sommes rendu compte qu'il serait intéressant d'implémenter certaines nouvelles fonctionnalités dans notre application. Même si nous n'avons pas eu le temps de le faire, nous pouvons tout de même vous présenter ces quelques idées.

Tout d'abord, l'ajout de nouvelles actions aux bots peut permettre de rendre leur comportement de plus en plus semblable à celui d'un humain et donc de les rendre moins prévisibles pour un véritable joueur. Par exemple, ces actions pourraient leur donner la possibilité d'effectuer des mouvements de contournement, de les faire interagir avec leur environnement en utilisant des objets pour se créer des couverts, bloquer des chemins ou en utilisant des coéquipiers comme bouclier lors des phases de feu adverse. Les possibilités sont nombreuses et elles peuvent apporter un véritable plus à l'expérience de jeu.

Dans cette optique, nous avions également commencé à créer un esprit d'équipe pour nos bots, avec une intelligence artificielle de groupe, afin que ceux-ci se déplacent et jouent réellement en totale coordination et qu'ils puissent mettre en place une stratégie commune. Ceci leur permettrait par exemple de contourner l'adversaire pendant que d'autres membres de l'équipe font face aux ennemis.

Pour finir, il serait également intéressant de mettre en place un apprentissage pour nos bots, afin qu'ils apprennent eux-mêmes de leur erreurs et qu'ils puissent ainsi mettre en place des stratégies efficaces. Mais c'est une phase longue à mettre en place, et nous avons manqué de temps pour en venir à bout.

## 5.4. Conclusion

Au cours de ce projet, nous avons proposé une solution aux objectifs qui étaient fixés : la création d'un nouveau mod nous permettant d'inclure nos propres bots dans le jeu et de charger des cartes personnalisées, et nous pensons avoir réussi à implémenter une intelligence artificielle correcte, avec des bots capables de simuler la plupart des actions nécessaires au jeu, en se mettant à couvert, changeant d'armes, se soignant, recherchant l'ennemi, ...

De plus, l'utilisation d'un planificateur nous a permis d'explorer une nouvelle voie intéressante qui commence tout juste à se développer dans le monde du jeu vidéo. Celui-ci étant trop souvent limité à de l'intelligence artificielle basée sur des machines à états finis.

D'un point de vue individuel, cette expérience fut vraiment enrichissante, aussi bien au niveau du travail en groupe qu'au niveau de l'apprentissage d'un nouveau langage, l'UnrealScript. De plus, ce projet fut une excellente expérience dans le domaine de l'intelligence artificielle, et le fait d'utiliser un moteur de jeu particulièrement connu, aussi bien dans le milieu professionnel que dans le milieu amateur, est un énorme avantage pour nous.

En résumé, ce projet nous a apporté un véritable plus et nous sommes heureux de l'avoir mené à bien.

## Annexes : Sommaire

# Annexes : Manuel d'utilisation

Pour pouvoir utiliser nos travaux sur votre machine, un certain nombre d'étapes sont nécessaires.

## • Fichiers et répertoires d'installation :

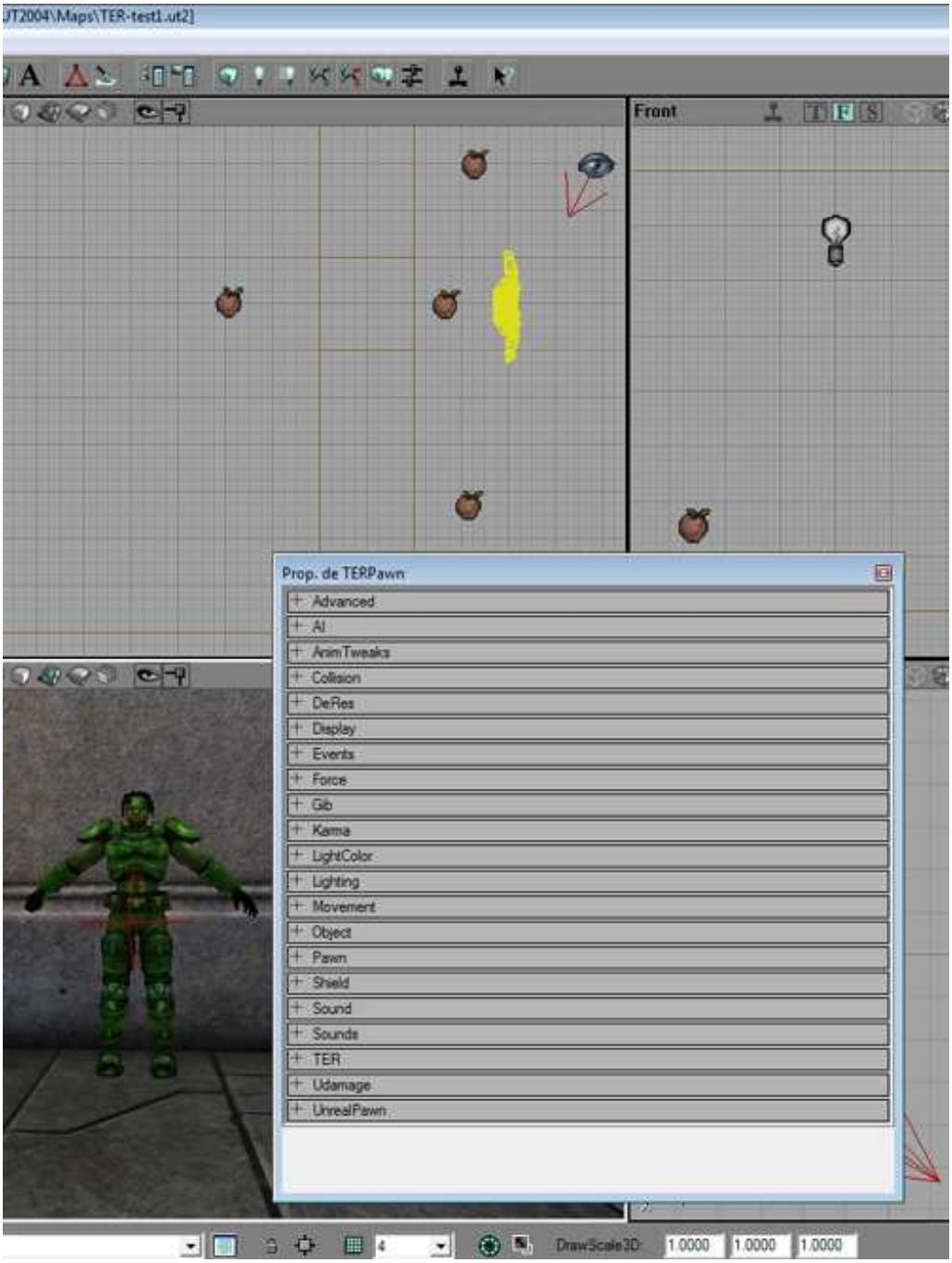
- Pour commencer, il vous faut installer le jeu Unreal Tournament 2004.
- Ensuite, il faut copier le dossier « TER » contenant le dossier « Classes » à la racine du répertoire d'installation du jeu Unreal.
- Il vous faut également copier les deux fichiers « TER-test1.ut2 » et « TER-test2.ut2 » dans le dossier « Maps » situé à la racine du répertoire d'installation du jeu Unreal.

## • Configurer les PNJ :

Pour simplifier les choses, nos cartes de tests sont déjà configurées, puisqu'elles contiennent déjà des PNJ placés. Ces PNJ sont également configurés selon différents comportements pour que vous puissiez les tester directement.

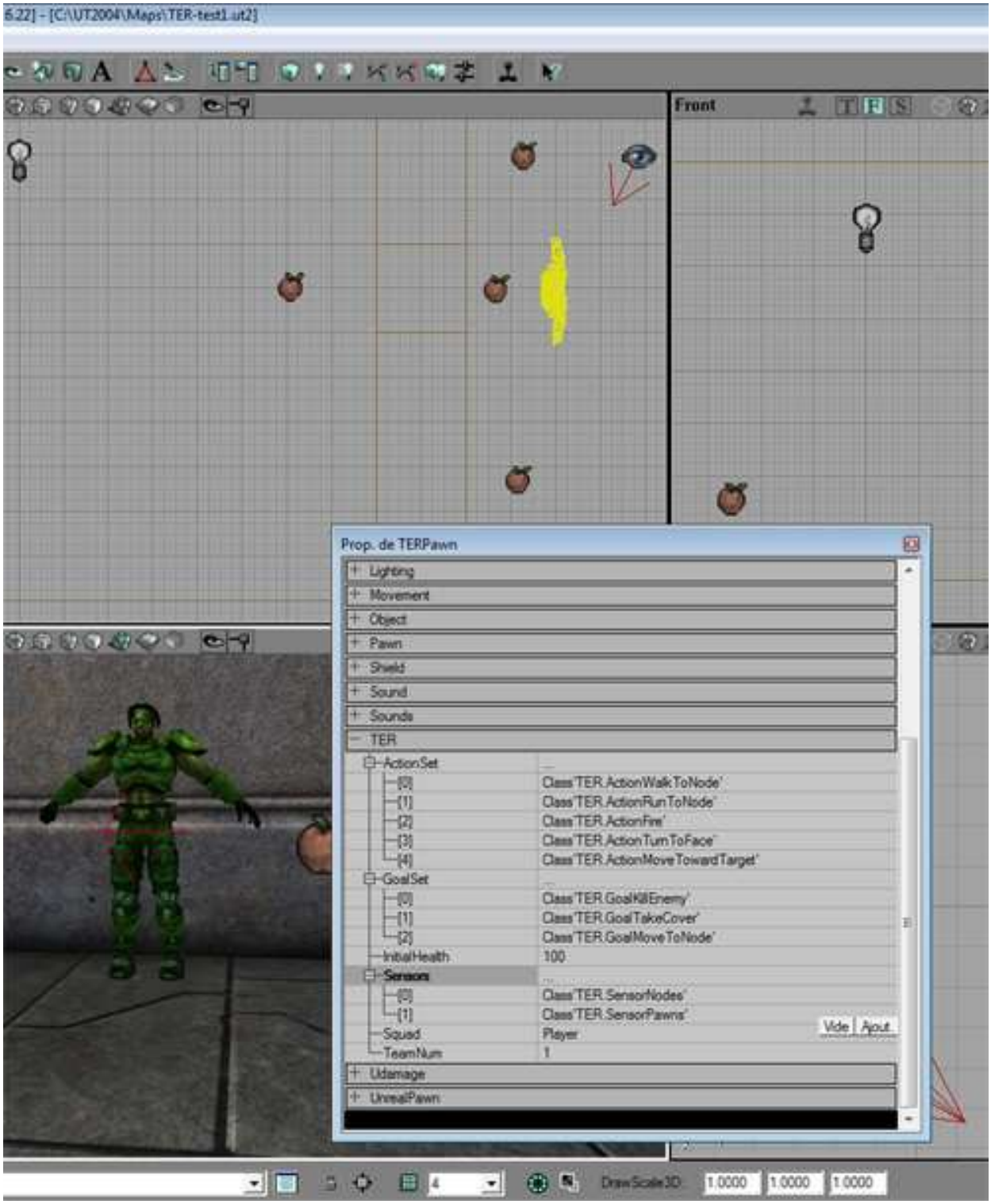
Toutefois, si vous souhaitez les configurer vous-même, il vous faut ouvrir l'éditeur du jeu, l'UnrealEd, puis charger une carte de jeu.

Par exemple : Fichier → Ouvrir → « TER-test2.ut2 »
Pour pouvoir configurer le comportement des bots, il vous faut ouvrir ses propriétés en double-cliquant dessus.
Vous obtenez alors un écran similaire à celui-ci :




Il vous faut alors ouvrir l'onglet « TER » et choisir les actions de votre bot dans la partie « ActionSet », ainsi que les buts dans la partie « GoalSet ». Vous pouvez également paramétrer ses points de vie, ...

Nous vous conseillons de les paramétrer comme ceci :




Comme vous le voyez, il faut également indiquer à quelle équipe le bot doit appartenir, l'équipe 0 ou bien l'équipe 1.

Enfin, pour pouvoir se déplacer sur la carte de jeu, il vous faut également placer des PathNodes. Pour cela, effectuez un clique droit sur la carte de jeu, puis choisissez « add Path Node here ».

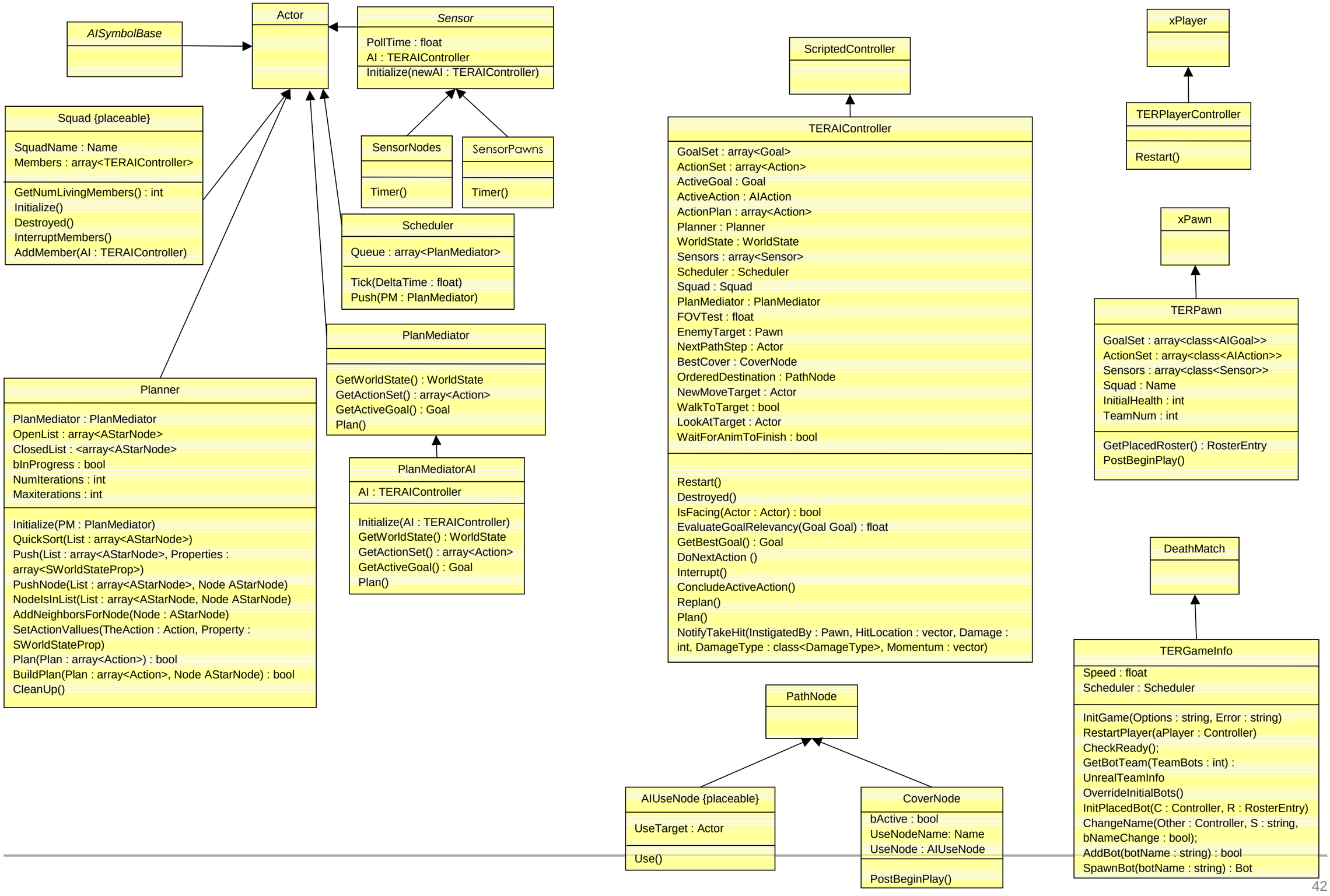
Sauvegardez maintenant votre carte, cliquez sur l'option « Build All » du menu « Build », puis sur « Play Level » de ce même menu pour tester votre configuration.

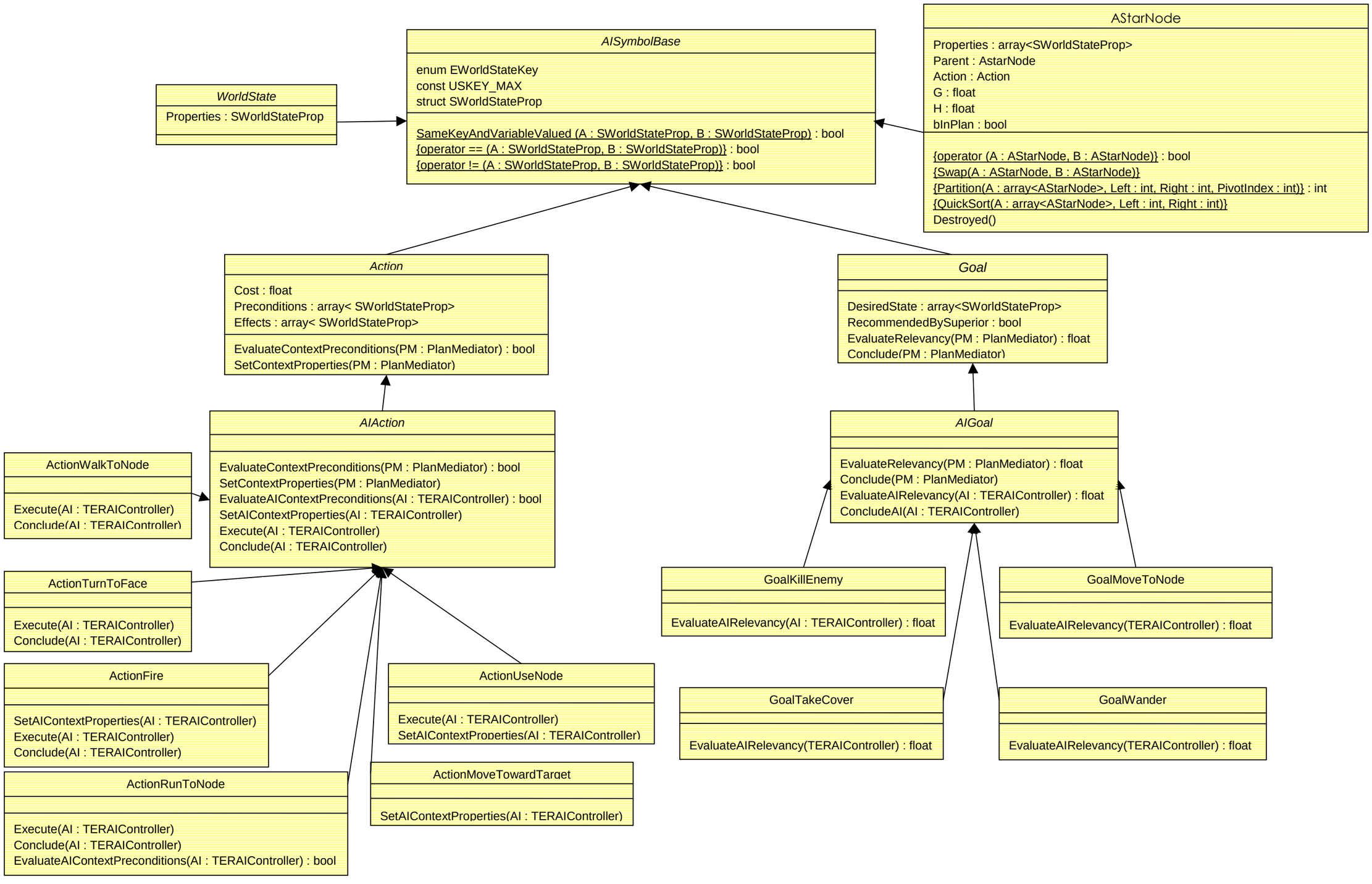
## Annexes : Webographie

- http://fr.wikipedia.org/wiki/Accueil
- http://www.unreal-design.com/
- http://wiki.beyondunreal.com/
- http://jeux.developpez.com/bibliotheques/
- http://www.gamedev.net/
- http://forums.epicgames.com/
- http://www.itu.dk/people/peder/
- http://giik.net/unrealpi/index.php3

## Annexes : Diagramme des classes

Ci-dessous, vous trouverez le diagramme de classes de notre projet. Celui-ci étant relativement imposant, nous avons du le séparer en deux parties pour plus de lisibilité. L'explication des classes principales se trouve dans la partie 2 de notre rapport, page 16.


AISymbolBase
Actor
Sensor
PollTime : float
AI : TERAIController
Initialize(newAI : TERAIController)
Squad {placeable}
SquadName : Name
Members : array<TERAIController>
GetNumLivingMembers() : int
Initialize()
Destroyed()
InterruptMembers()
AddMember(AI : TERAIController)
SensorNodes
Timer()
SensorPawns
Timer()
Scheduler
Queue : array<PlanMediator>
Tick(DeltaTime : float)
Push(PM : PlanMediator)
PlanMediator
GetWorldState() : WorldState
GetActionSet() : array<Action>
GetActiveGoal() : Goal
Plan()
Planner
PlanMediator : PlanMediator
OpenList : array<AStarNode>
ClosedList : <array<AStarNode>
bInProgress : bool
NumIterations : int
Maxiterations : int
Initialize(PM : PlanMediator)
QuickSort(List : array<AStarNode>)
Push(List : array<AStarNode>, Properties : array<SWorldStateProp>)
PushNode(List : array<AStarNode>, Node AStarNode)
NodeIsInList(List : array<AStarNode, Node AStarNode)
AddNeighborsForNode(Node : AStarNode)
SetActionVallues(TheAction : Action, Property : SWorldStateProp)
Plan(Plan : array<Action>) : bool
BuildPlan(Plan : array<Action>, Node AStarNode) : bool
CleanUp()
PlanMediatorAI
AI : TERAIController
Initialize(AI : TERAIController)
GetWorldState() : WorldState
GetActionSet() : array<Action>
GetActiveGoal() : Goal
Plan()
ScriptedController
TERAIController
GoalSet : array<Goal>
ActionSet : array<Action>
ActiveGoal : Goal
ActiveAction : AIAction
ActionPlan : array<Action>
Planner : Planner
WorldState : WorldState
Sensors : array<Sensor>
Scheduler : Scheduler
Squad : Squad
PlanMediator : PlanMediator
FOVTest : float
EnemyTarget : Pawn
NextPathStep : Actor
BestCover : CoverNode
OrderedDestination : PathNode
NewMoveTarget : Actor
WalkToTarget : bool
LookAtTarget : Actor
WaitForAnimToFinish : bool
Restart()
Destroyed()
IsFacing(Actor : Actor) : bool
EvaluateGoalRelevancy(Goal Goal) : float
GetBestGoal() : Goal
DoNextAction ()
Interrupt()
ConcludeActiveAction()
Replan()
Plan()
NotifyTakeHit(InstigatedBy : Pawn, HitLocation : vector, Damage : int, DamageType : class<DamageType>, Momentum : vector)
xPlayer
TERPlayerController
Restart()
xPawn
TERPawn
GoalSet : array<class<AIGoal>>
ActionSet : array<class<AIAction>>
Sensors : array<class<Sensor>>
Squad : Name
InitialHealth : int
TeamNum : int
GetPlacedRoster() : RosterEntry
PostBeginPlay()
PathNode
AIUseNode {placeable}
UseTarget : Actor
Use()
CoverNode
bActive : bool
UseNodeName: Name
UseNode : AIUseNode
PostBeginPlay()
DeathMatch
TERGameInfo
Speed : float
Scheduler : Scheduler
InitGame(Options : string, Error : string)
RestartPlayer(aPlayer : Controller)
CheckReady();
GetBotTeam(TeamBots : int) : UnrealTeamInfo
OverrideInitialBots()
InitPlacedBot(C : Controller, R : RosterEntry)
ChangeName(Other : Controller, S : string, bNameChange : bool);
AddBot(botName : string) : bool
SpawnBot(botName : string) : Bot


AStarNode
Properties : array<SWorldStateProp>
Parent : AstarNode
Action : Action
G : float
H : float
bInPlan : bool
{operator (A : AStarNode, B : AStarNode)} : bool
{Swap(A : AStarNode, B : AStarNode)}
{Partition(A : array<AStarNode>, Left : int, Right : int, PivotIndex : int)} : int
{QuickSort(A : array<AStarNode>, Left : int, Right : int)}
Destroyed()
AISymbolBase
enum EWorldStateKey
const USKEY_MAX
struct SWorldStateProp
SameKeyAndVariableValued (A : SWorldStateProp, B : SWorldStateProp) : bool
{operator == (A : SWorldStateProp, B : SWorldStateProp)} : bool
{operator != (A : SWorldStateProp, B : SWorldStateProp)} : bool
WorldState
Properties : SWorldStateProp
Action
Cost : float
Preconditions : array< SWorldStateProp>
Effects : array< SWorldStateProp>
EvaluateContextPreconditions(PM : PlanMediator) : bool
SetContextProperties(PM : PlanMediator)
Goal
DesiredState : array<SWorldStateProp>
RecommendedBySuperior : bool
EvaluateRelevancy(PM : PlanMediator) : float
Conclude(PM : PlanMediator)
AIAction
EvaluateContextPreconditions(PM : PlanMediator) : bool
SetContextProperties(PM : PlanMediator)
EvaluateAIContextPreconditions(AI : TERAIController) : bool
SetAIContextProperties(AI : TERAIController)
Execute(AI : TERAIController)
Conclude(AI : TERAIController)
AIGoal
EvaluateRelevancy(PM : PlanMediator) : float
Conclude(PM : PlanMediator)
EvaluateAIRelevancy(AI : TERAIController) : float
ConcludeAI(AI : TERAIController)
ActionWalkToNode
Execute(AI : TERAIController)
Conclude(AI : TERAIController)
ActionTurnToFace
Execute(AI : TERAIController)
Conclude(AI : TERAIController)
ActionFire
SetAIContextProperties(AI : TERAIController)
Execute(AI : TERAIController)
Conclude(AI : TERAIController)
ActionUseNode
Execute(AI : TERAIController)
SetAIContextProperties(AI : TERAIController)
ActionRunToNode
Execute(AI : TERAIController)
Conclude(AI : TERAIController)
EvaluateAIContextPreconditions(AI : TERAIController) : bool
ActionMoveTowardTarget
SetAIContextProperties(AI : TERAIController)
GoalKillEnemy
EvaluateAIRelevancy(AI : TERAIController) : float
GoalMoveToNode
EvaluateAIRelevancy(TERAIController) : float
GoalTakeCover
EvaluateAIRelevancy(TERAIController) : float
GoalWander
EvaluateAIRelevancy(TERAIController) : float

## Annexes : Extraits de code

Dans cette partie, nous allons vous présenter quelques extraits de code choisis, afin d'étayer nos propos et peut être ainsi clarifier certains détails.

Nous allons tout d'abord vous présenter les nœuds utilisés pour le planificateur.

```
// AStarNode.uc

// Noeud de AStar pour le 'Planner'.
// Un noeud dans le graphe est un 'état du monde'.

class AStarNode extends AISymbolBase;

// Les propiétés qui doivent être satisfaites.
var array<SWorldStateProp>   Properties;
// Référence directe au noeud suivant (utile pour construire le plan)
var AStarNode                        Parent;
// Action correspondante au noeud Parent.
var   Action                                 Action;
// Le coût (fonction g() de AStar)
var float                            G;
// L'heuristique (fonction h() de AStar)
var float                            H;
// Pour savoir si un noeud fait partie ou non du plan (utile lors de la
destruction)
var bool                             bInPlan;

function DEBUG_Log()
{
      local int i;
      Log( "Node: "$Name );
      Log( "Properties:" );
      for( i = 0; i < Properties.Length; ++i )
      {
            DEBUG_LogProperty( Properties[i] );
      }
      if( Action != None )
      {
            Log( "Action: "$Action.Name );
      }
      Log( "A* Value: "$(G+H)$" = "$G$" + "$H );
      Log( "----------" );
}

static final operator(24) bool < ( AStarNode A, AStarNode B )
{
      return ( A.G + A.H < B.G + B.H );
}

// Swap -- utilisée par Partition
// Echange deux valeurs...
static final function Swap( out AStarNode A, out AStarNode B )
{
      local AStarNode Temp;
```

```
	Temp = A;
	A = B;
	B = Temp;
}

// Partition -- utilisée par QuickSort
// Retourne un nouveau pivot.
static final function int Partition( out array<AStarNode> A, int Left, int
Right, int PivotIndex )
{
	local AStarNode   PivotValue;
	local int         i;
	local int         j;

	PivotValue = A[ PivotIndex ];
	Swap( A[ PivotIndex ], A[ Right ] );
	j = Left;
	for( i = Left; i < Right; ++i )
	{
		if( A[ i ] < PivotValue )
		{
			Swap( A[ i ], A[ j ] );
			j++;
		}
	}
	Swap( A[ Right ], A[ j ] );

	return j;
}

// Tri Rapide.
static final function QuickSort( out array<AStarNode> A, int Left, int
Right )
{
	local int   PivotIndex;
	if( Right > Left )
	{
		// Point au milieu...
		PivotIndex = ( Right + Left ) >> 1;
		PivotIndex = Partition( A, Left, Right, PivotIndex );
		QuickSort( A, Left, PivotIndex - 1 );
		QuickSort( A, PivotIndex + 1, Right );
	}
}

function Destroyed()
{
	if( !bInPlan && Action != None )
	{
		// Nettoyer les actions unitilisées (mises par le 'Planner')
		Action.Destroy();
	}

	Super.Destroyed();
}

defaultproperties{}
```

Nous allons maintenant vous présenter notre ordonnanceur qui permet aux bots de savoir quand est-ce qu'ils peuvent concevoir leur plan.

```
// Scheduler.uc

// L'ordonnanceur est une file (FIFO) de TERAIControllers.
// On les appelle quand c'est à eux de recevoir un plan.

class Scheduler extends Actor;

var array<PlanMediator> Queue;

function Tick( float DeltaTime )
{
	if( Queue.Length > 0 )
	{
		if( Queue[0] != None )
		{
			Queue[0].Plan();
		}
		Queue.Remove( 0, 1 );   // "Pop" la "queue"
	}
}

function Push( PlanMediator PM )
{
	if( PM != None )
	{
		Queue[ Queue.Length ] = PM;   // "Push" sur la "queue"
	}
}

defaultproperties
{
	bHidden=true
}
```

Voici maintenant l'exemple d'une action réalisable par les bots. Comme vous pouvez le voir et comme nous l'avons expliqué, nos actions sont courtes dans leur implémentation.

Il s'agit ici de l'action qui permet à nos bots de faire feu.

```
// ActionFire.uc

// Action de tirer sur une cible.

class ActionFire extends AIAction;

function SetAIContextProperties( TERAIController AI )
{
      Preconditions[1].aValue = AI.EnemyTarget;
}

function Execute( TERAIController AI )
{
      local Actor            TraceActor;
      local TERPawn         P;
      local TERPawn         AIPawn;
      local Vector           HitLocation;
      local Vector           HitNormal;

      AIPawn = TERPawn( AI.Pawn );


    // DEBUG
      //Log( "Executing ActionFire", 'TER' );


      // On vérifie tout d'abord que l'ennemi (EnemyTarget) est visible.
      if( AI.CanSee( AI.EnemyTarget ) )
      {
            // On vérifie ensuite qu'on ne risque pas de tirer sur un
allié.
        TraceActor = AI.Trace( HitLocation, HitNormal,
AI.EnemyTarget.Location - AIPawn.Location, AIPawn.Location, true );
            P = TERPawn( TraceActor );

        // On ignore tout les autres Actor.
        // On vérifie juste que l'on est pas dans la même équipe (TeamNum).
            if( P == None || P.TeamNum != AIPawn.TeamNum )
            {
                  AI.LookAtTarget = AI.EnemyTarget;
                  AI.Pawn.Weapon.BotFire( false );
                  AI.WaitForAnimToFinish = false;
                  AI.SetTimer( 1.f, false );
            }
      }
      AI.GotoState( 'TER_Animate' );
}
```

```
function Conclude( TERAIController AI )
{
	AI.StopFiring();
}

defaultproperties
{
	Cost=1.f;
	Preconditions(0)=(Key=WSKEY_CanSeeTarget,bValue=true)
	Preconditions(1)=(Key=WSKEY_FacingActor,bVariableValue=true)
	Effects(0)=(Key=WSKEY_TargetIsDead,bValue=true)
}
```

Et enfin nous terminerons en vous présentant un objectif à atteindre pour nos bots : se déplacer vers un nœud.

```
// GoalMoveToNode.uc

// Le but ici est de privilégier un noeud où le bot sera à couvert
// quand celui-ci n'a pas de cible.

class GoalMoveToNode extends AIGoal;

function float EvaluateAIRelevancy( TERAIController AI )
{
	DesiredState[0].aValue = AI.OrderedDestination;

	if( AI.OrderedDestination != None &&
AI.WorldState.Properties[2].aValue != AI.OrderedDestination )
	{
		return 0.1f;
	}
	return 0.f;
}

defaultproperties
{
	DesiredState(0)=(Key=WSKEY_AtNode,bVariableValue=true)
}
```