UNIVERSITÉ DE
SHERBROOKE

Projet IFT 611-729

# Conception temps réel

Julien Levarlet
Marius Pallard
Thomas Roux-Croteau
Thouria Ben Haddi

*Encadrant :*
M.Patrice Roy

12 avril 2022

# 1 Livrable 02

## 1.1 Travail effectué

L'objectif principal du projet consiste à implémenter une intelligence artificielle de jeu-vidéo capable de réagir aux actions du joueur sous la forme d'un ou plusieurs ennemis.
Le travail effectué a permis d'obtenir un jeu où l'IA est capable de traquer le joueur pour l'éliminer de manière polyvalente grâce à un système d'attaque et de points de vie, un système de jeu s'adaptant aux événements de la partie, des caméras pour plus d'immersion pour le joueur et mieux observer la scène, une multitude d'environnements différents ainsi qu'un algorithme génétique permettant à l'IA de s'entraîner et d'avoir un comportement s'adaptant aux actions du joueur pour lui poser un challenge.

Le projet est disponible sur GitHub à l'adresse suivant : `https://github.com/julien-levarlet/Projet_STR`.

## 1.2 Organisation du projet

Deux types d'organisation du code sont mise en place en parallèle, il y a le cas d'une partie classique et de l'entraînement d'algorithme génétique.
Remarque : `AstarAgent` n'est pas parfaitement intégrée au `GameManager`, mais une scène met en place un exemple de fonctionnement de cet algorithme (sans implémentation du système de partie)

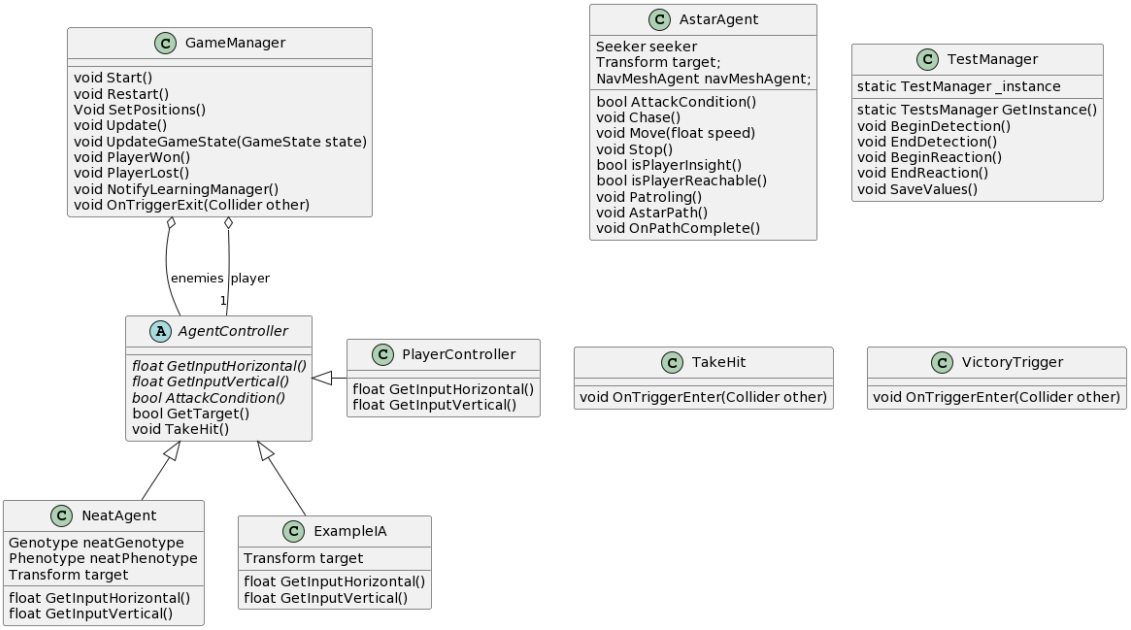


FIGURE 1.1 – Diagramme de classe de notre projet

Dans le cas de l'entrainement, le but est de retirer tous les comportements superflus de la partie (multiples cameras, écrans de victoire, etc), dans le but d'avoir un entrainement le plus efficace possible.

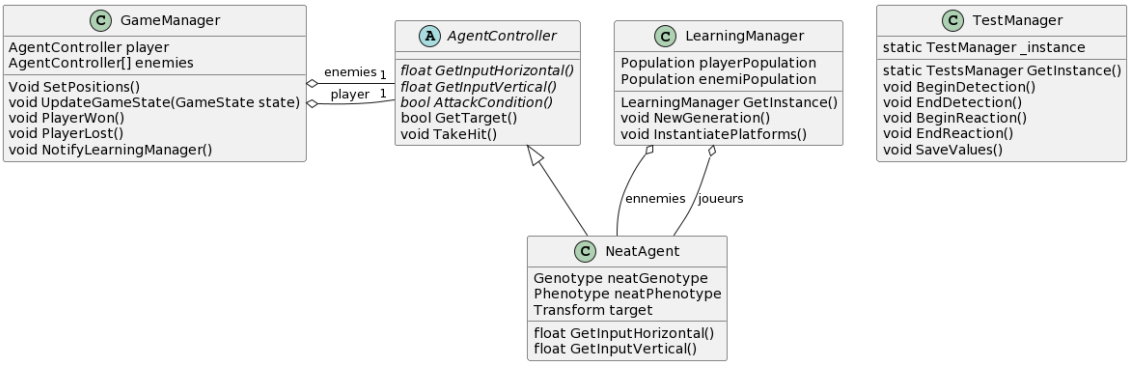


FIGURE 1.2 – Diagramme de classe d'entrainement

### 1.2.1 Fabrication de scènes et d'effets pour le jeu

L'environnement dans lequel se passe le jeu est constitué de plusieurs arènes. Ces arènes sont conçues pour avoir des obstacles différents d'une scène à l'autre afin de tester le comportement des IAs dans diverses situations et de pouvoir proposer plus de gameplay pour le joueur (voir L01 pour plus de détail, la composition des arènes n'a pas vraiment changé).

Le système de jeu marche ainsi : à chaque partie un agent joueur et deux agents ennemis apparaissent dans une arène quelconque.
Le but du joueur est d'atteindre une zone sur l'arène qui correspond à un point de victoire, il n'a qu'à la toucher pour gagner et déclencher l'écran de victoire.
Ce point de victoire apparaît en même temps que les agents, ils sont tous répartis sur 4 points sur la scène de manière symétrique, mais le point d'apparition sur lequel un agent ou le point de victoire apparaît est aléatoire, cela implique que la répartition est différente à chaque partie.

Le but du joueur est donc d'atteindre ce point, le but des ennemis quant à eux est d'éliminer le joueur avant qu'il atteigne cet objectif.

Le déroulement de la partie est géré à l'aide d'un script `GameManager` que l'on applique sur chaque scène. Le `GameManager` fonctionne selon un système d'état.
Chaque état correspond à une situation de la partie et appelle des fonctions diverses à chaque mise-à-jour.
Le changement d'état peut être appelé par un autre élément du jeu. Par exemple, lorsque l'élément point de victoire détecte qu'il est en contact avec l'agent du joueur, il appelle la fonction `UpdateGameState()` en passant en paramètre l'état "Victory". L'état "Victory" fait en sorte lors de la mise-à-jour que la fonction `PlayerWon()` est appelée, ce qui déclenche l'écran de victoire.

Le joueur et les ennemis sont des agents, ils ont accès aux mêmes propriétés et mouvements, à la différence que l'un est contrôlé par le joueur et les autres par une IA.
Un agent peut se déplacer en allant de l'avant, en reculant et en pivotant vers la gauche ou vers la droite pour tourner. Pour le joueur, ce sont les inputs de flèches sur le clavier qui effectuent ces mouvements

Un agent possède une attaque permettant d'infliger des dégâts à un autre agent. L'attaque peut s'apparenter à un coup latéral d'une épée et frappe en face de l'agent. L'attaque est déclenchée par un clic gauche de souris pour le joueur et pour l'ennemi, l'attaque se déclenche automatiquement lorsque le joueur se trouve à une certaine distance de lui.

Les attaques et les dégâts reçus sont gérés à l'aide d'un système de boîtes de collisions. Chaque agent possède une Hurtbox autour de son corps et une Hitbox pour son attaque.
Lorsque la Hitbox d'une attaque entre en contact avec une Hurtbox, l'agent de la Hurtbox correspondante prend un point de dégât et perd une vie. Si un agent perd la totalité de ses vies, il est alors éliminé de la partie (ce qui consiste en sa désactivation et il disparaît de la scène en cours).
Le joueur peut donc éliminer les ennemis avant d'atteindre son objectif, s'il se fait attaquer et meurt par contre, la partie se termine par une défaite.

L'attaque est une action en mouvement, il aura donc fallu la gérer avec le système d'animation d'Unity. Le mouvement fonctionne ainsi : à l'état de repos la Hitbox n'est pas présente, son échelle est à 0. Lorsque l'événement de l'attaque se déclenche, la Hitbox apparaît et suit la rotation d'un repère placé sur le joueur. L'agent doit finir l'animation de l'attaque pour en relancer une nouvelle, il ne peut donc pas en faire plusieurs simultanément.

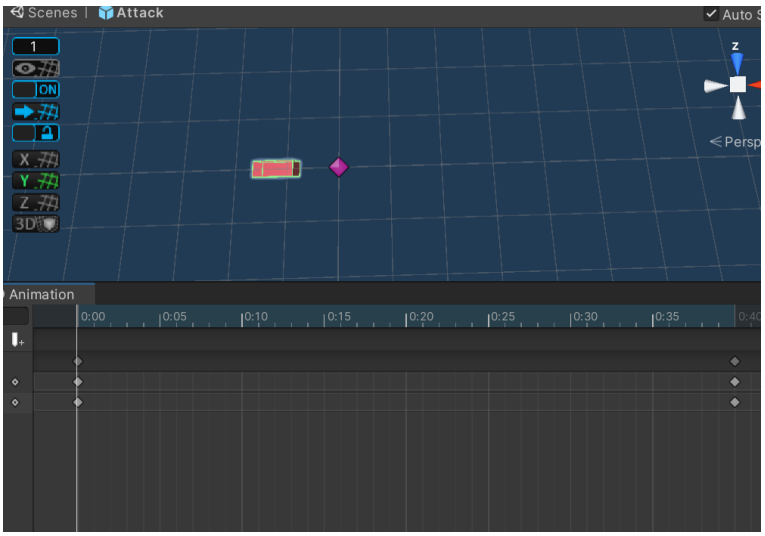


FIGURE 1.3 – Début de l'attaque de l'agent

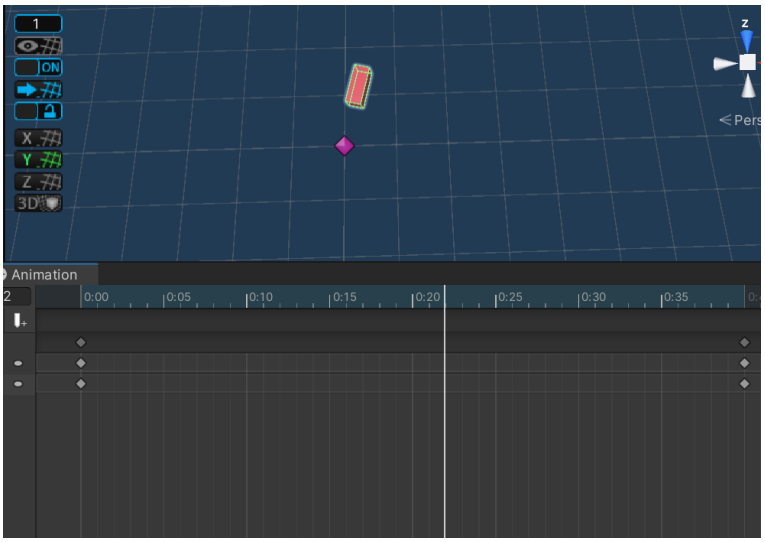


FIGURE 1.4 – Déroulement de l'attaque de l'agent

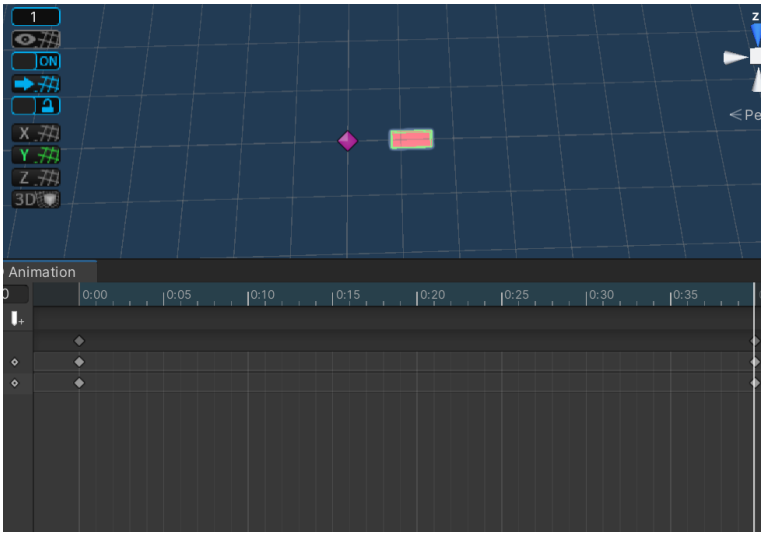


FIGURE 1.5 – Fin de l'attaque de l'agent

Ce système d'attaque combiné aux déplacements est la base du gameplay du jeu et c'est sur cela que le système d'intelligence artificielle va se baser pour tenter d'éliminer le joueur de la manière la plus efficace.

### 1.2.2 Caméras

Un ensemble de caméras est attribué à chaque scène, certaines attachées à des agents (qui cessent donc d'exister à la mort/disparition de ceux-ci) et d'autres plus en hauteur telle que la caméra type de surveillance ou la caméra type jeu de stratégie.
Ces caméras permettent de bien comprendre ce qui se passe en jeu quelle que soit l'arène. (Pour plus de détails voir L01, les caméras n'ont pas vraiment changé)

### 1.2.3 Mouvement des IAs

#### IA de poursuite

La première implémentation de pathfinding pour les ennemis se basait sur des méthodes simples consistant à vérifier si le joueur se trouvait à portée de l'ennemi à chaque appel de la fonction `Update()`. Le script *AstarAgent.cs* se décompose en plusieurs fonctions avec notamment 3 fonctions principales. La première permet de patrouiller aléatoirement sur l'arène, la seconde de vérifier à l'aide de Raycast si le joueur se trouve dans le champ de vision de l'IA ou non, et enfin la dernière de prendre en chasse le joueur afin de pouvoir l'attaquer une fois à portée.

La fonction de chasse étant élémentaire et simple, nous souhaitions implémenter une méthode plus élaborée abordée dans les cours d'Intelligence Artificielle à l'université. C'est dans cette optique que nous nous sommes tournés vers l'algorithme A*. Ainsi, A* nous permet d'avoir le chemin le plus court entre le joueur et l'ennemi pour le prendre en chasse. Et ce en évitant les obstacles. De plus, un avantage de cet algorithme est qu'il nous assure de produire une solution en sortie.

Nous avons donc tenté l'implémentation de l'algorithme avec notamment les scripts s'occupant de la gestion de la grille et la définition des nœuds sur cette dernière. On trouve deux listes dans l'algorithme, la première, la liste ouverte, contient les nœuds déjà étudiés jusqu'au nœud actuel, la seconde liste, liste fermée, contiendra tous les nœuds qui durant la recherche de chemin ont été considérés comme admissibles pour le chemin solution final. L'implémentation de l'algorithme en lui-même consistait à vérifier l'admissibilité, pour le nœud actuel où se trouvait l'ennemi, d'un nœud voisin sur lequel avancer afin de minimiser la distance entre le joueur et l'ennemi. Enfin, cette itération était répétée jusqu'à atteindre l'objectif, soit le joueur dans notre cas.

Nous avons rencontré quelques problèmes lors de la mise en place d'A*, notamment la mise en place de la grille de nœuds. La définition des nœuds sur l'arène nous a bloqué et ne nous a pas permis de tester l'implémentation de l'algorithme. Le résultat ne fonctionnant pas, nous avons donc utilisé un asset [1] qui implémente de façon complète l'algorithme afin de pouvoir l'utiliser dans notre projet et de parer nos difficultés. L'asset nous permet d'utiliser une grille adaptable à l'arène considérée et d'exécuter l'algorithme sur la cible de notre choix. Nous avons donc utilisé le code mis à disposition pour l'implémentation même d'A* et l'avons adapté à notre scénario (dans le script `AstarAgent.cs`) avec notamment la phase de patrouille et de détection du joueur avant l'appel à A* pour prendre en chasse le joueur.

1. `https://arongranberg.com/astar/features`

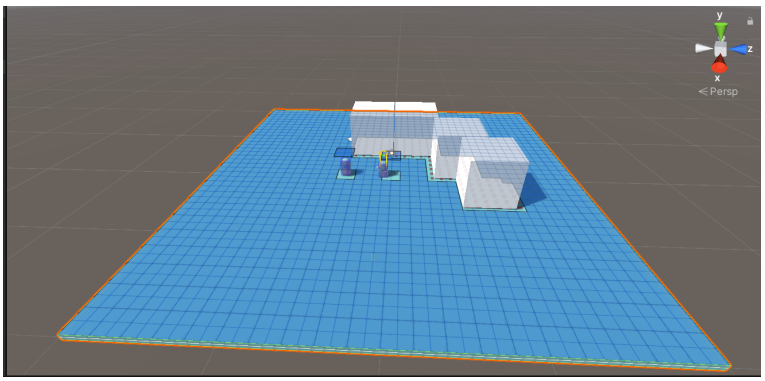


FIGURE 1.6 – Grille de nœuds pour A*

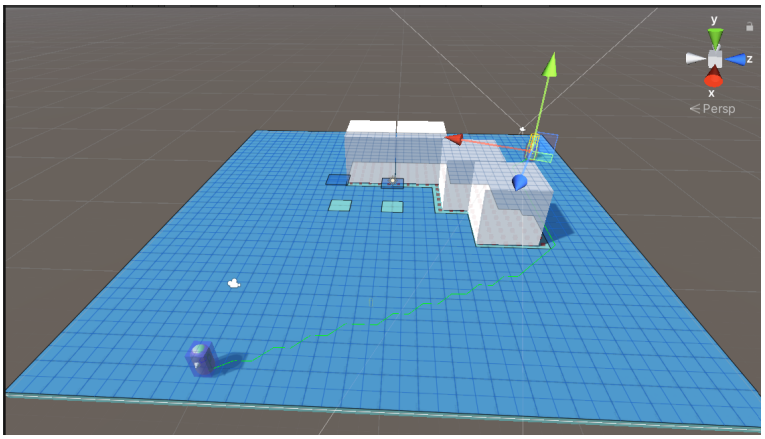


FIGURE 1.7 – Chemin déterminé par A*

L'IA A* reste cependant imparfaite puisqu'elle ne permet pas l'utilisation de notre `AgentController` qui impose d'utiliser des fonctions de déplacement avec des méthodes différentes de celles utilisées par notre IA A*. Le Controller permet notamment la gestion des points de vie des agents et est donc indispensable à la gestion d'une partie.

Enfin, l'implémentation d'A* nous servira afin de comparer les contraintes temps réel avec l'algorithme génétique implémenté par la suite et ne sera donc pas utilisable en jeu.

### IA par algorithme génétique

L'algorithme que nous avons choisi d'utiliser en complément de ce que l'on a fait avec A* est l'algorithme NEAT (Neuroevolution of augmenting topologies).

Cet algorithme se base sur l'utilisation d'un génotype et d'un phénotype. Le génotype contient toutes les caractéristiques de notre IA qui seront utilisées dans les étapes de l'algorithme génétique. Le phénotype est un réseau de neurone, dont la taille et les poids sont déterminés par le génotype. À chaque génération, les meilleurs individus sont sélectionnés selon des critères que nous présenterons dans la suite. Ensuite, les caractéristiques des meilleurs individus sont croisées, puis mutées, afin de créer une nouvelle génération.

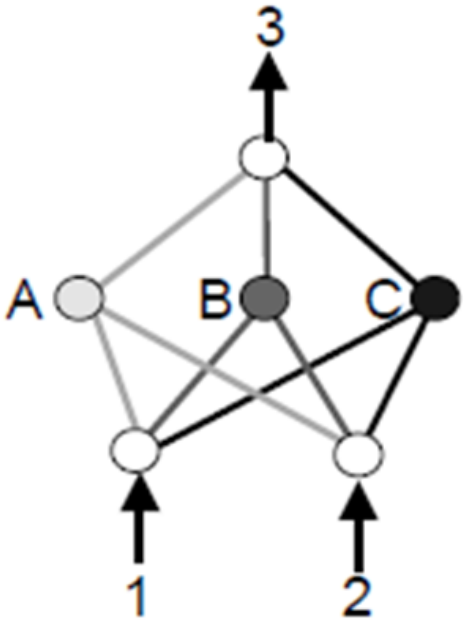


FIGURE 1.8 – Exemple de réseau produit par NEAT

Dans notre cas, le réseau de neurone à 7 entrées :

- son angle par rapport au monde
- sa position sur la scène (2D)
- la position du joueur (2D)
- la position du point de victoire

Et il a trois sorties :

- sa rotation (entre -1 et 1)
- sa vitesse (entre -1 et 1)
- sa décision d'attaque (entre 0 et 1, supérieur à 0.5 correspond à attaquer)

Pour entraîner l'IA de l'ennemi, nous avons procédé par l'utilisation d'un algorithme génétique. On l'a donc entraînée en lui donnant des points si elle effectuait certaines actions afin de lui insuffler un comportement.
Pour l'entraîner cependant, on n'a pas fait 10 000 parties où un humain joue contre elle, cela aurait été trop long, donc en parallèle de l'entraînement, on a entraîné une IA à imiter le comportement d'un joueur pour que l'ennemi puisse l'affronter et progresser grâce à cela.

Concrètement, nous générons 200 arènes faites avec différents prefab :

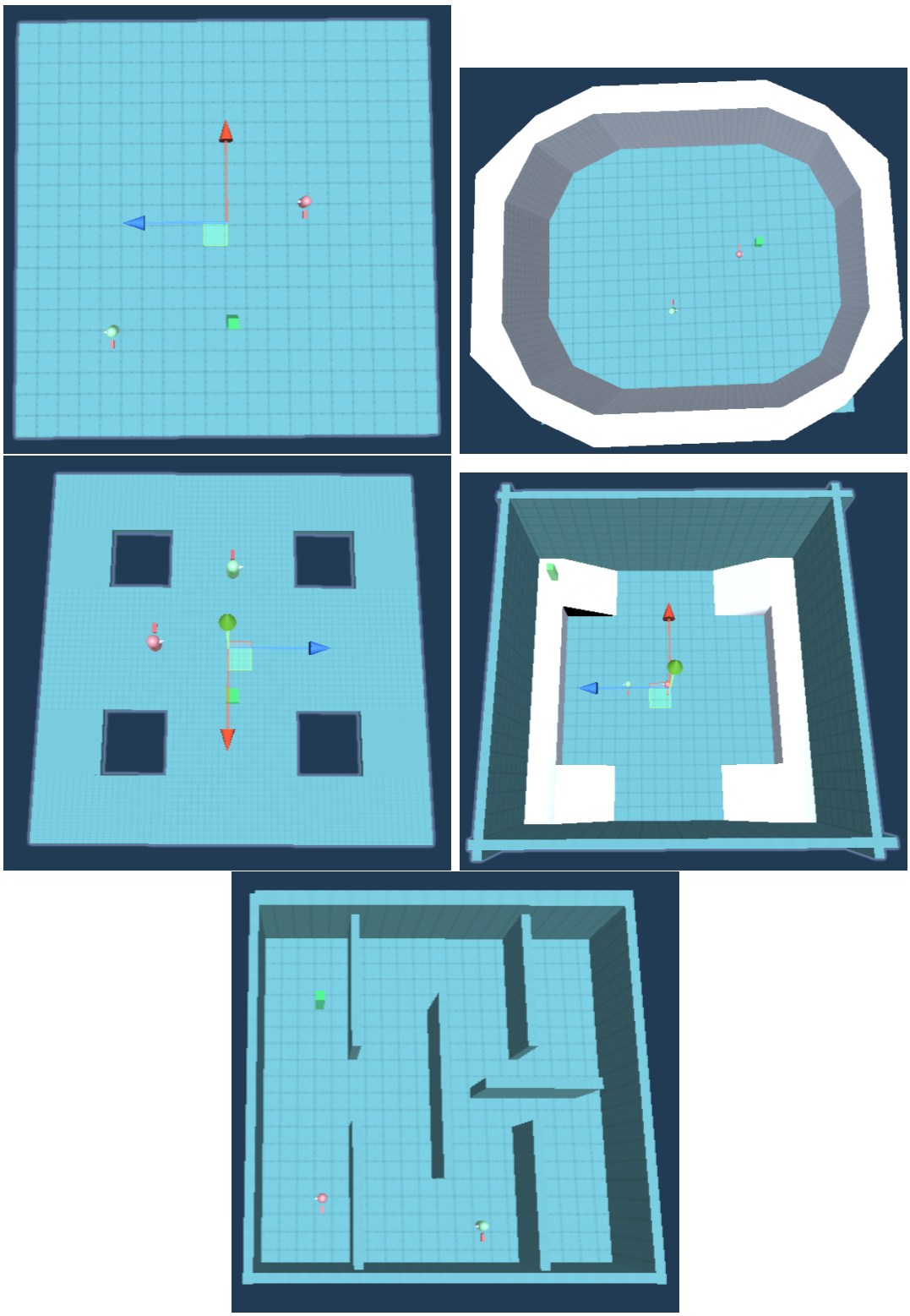

FIGURE 1.9 – Prefabs des plateformes possibles

Sur ces arènes, nous faisons dans un premier temps apparaitre à des localisations fixées le joueur, l'ennemi et la position à atteindre pour gagner.
Le but final étant que les IAs puissent généraliser et parvenir à voir le bon comportement, y compris avec une position de départ différente de celle habituelle.
Une fois l'entraînement suffisamment satisfaisant, le but serait d'entraîner les IA avec des positions initiales aléatoires.

Pour réaliser l'entrainement, on crée une génération d'IA, dans notre cas avec une population de 200 agents. À la première génération, les agents vont avoir des comportements "aléatoires" tout en étant plus ou moins récompensés, jusqu'à affiner leur conduite, grâce au système de récompense.

Dans le cas du joueur, les récompenses sont les suivantes :

- À chaque update, le joueur gagne une récompense de 10 fois l'écart entre sa distance initiale à l'objectif et sa distance actuelle à celui-ci.
  Cela a pour but de lui indiquer qu'il doit se rapprocher du point de victoire.
- Pour lui faire comprendre qu'il doit aller vite à l'objectif, on fait perdre 2 fois les points qu'il gagne grâce à l'écart de distance mentionné juste avant dans le cas où il gagne, sinon il ne perd rien.
- Si le joueur a gagné (à atteint le point de victoire), il reçoit une récompense de 100 000 points. C'est significativement plus grand que toutes les autres récompenses, car c'est son objectif principal.
- Qu'il ait perdu ou gagné, lorsque la partie actuelle se termine, le joueur gagne 50 points par ennemi tué pour lui faire comprendre qu'il doit tuer les ennemis rencontrés.

Et pour l'IA de l'agent ennemi :

- Comme pour l'agent du joueur, l'ennemi gagne des récompenses en fonction d'une distance : de son écart à l'objectif (comme pour le joueur) mais multiplié par 3. Le but est qu'il se rapproche de l'objectif afin de bloquer le joueur.
- Et la même récompense d'écart de distance par rapport au joueur cette fois et bien multiplié par 10. Pour que bien qu'il se rapproche de l'objectif, son objectif est avant tout d'atteindre le joueur.
- À chaque fois que le joueur perd une vie, l'ennemi gagne 20 000 points. Une grosse somme afin de lui signifier qu'il doit attaquer le joueur et l'éliminer
- Une récompense de 2 points par update pour insuffler à l'ennemi la volonté de survivre et de ne pas mourir.

On lance l'entraînement sur Unity et pour procéder plus rapidement, on accélère l'entraînement en modifiant le timescale (valeur modifiable gérée par Unity) en le rendant 5 fois plus rapide. Si cependant on veut regarder l'évolution du comportement des IAs, il suffit d'appuyer et de maintenir la touche A pour remettre la vitesse normale.
Une fois qu'on est satisfait, on peut enregistrer les entraînements des IAs avec la touche S.

On a donc entraîné plusieurs IA, une par niveau, afin que quand le joueur joue dans le niveau correspondant, il affronte l'IA la plus adaptée à le contrer.

### 1.2.4 Contraintes temps réel

Dans un premier temps, nous allons vérifier que nos contraintes sont bien vérifiées dans le cas de l'algorithme NEAT.

Notre contrainte de brièveté :
On souhaitait un temps de réaction de l'IA pertinent pour qu'il soit réactif face aux actions du joueur et assurer une fluidité et spontanéité de jeu.
Ainsi, l'IA devait être capable de repérer (dans son rayon de perception), le joueur dans un délai de temps inférieur à 273 ms (temps moyen de réaction d'un humain et d'agir en conséquence.
Pour vérifier cette contrainte, on a relevé quand est ce que le jeu détectait que l'ennemi et le joueur étaient séparés d'une distance inférieure ou égale au rayon de perception. Puis on a relevé quand est ce que l'IA détectait le joueur, à savoir quand est ce qu'on rentrait dans le script de détection de l'IA et on a mesuré le décalage suivant :

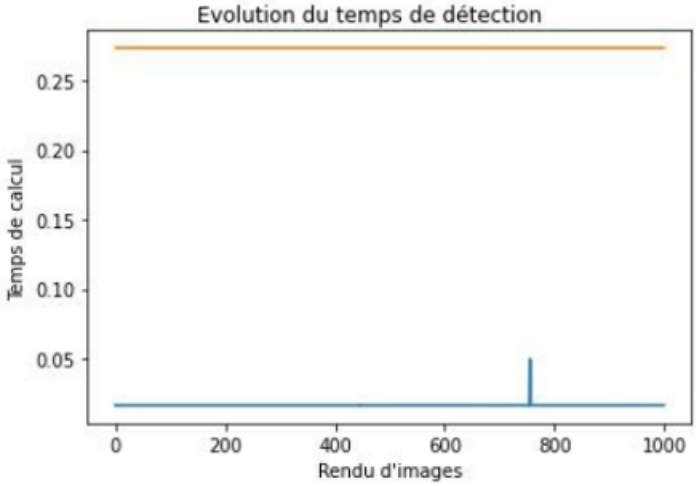


FIGURE 1.10 – Contrainte Algorithme NEAT

Comme on peut le voir, la détection est extrêmement régulière (mis de côté un petit pic) et reste toujours en dessous de 100ms.
C'est très correct et ça satisfait nos exigences.

Notre contrainte d'immédiateté :
On voulait qu'entre l'appui d'une touche clavier ou l'utilisation d'un périphérique (la souris) et l'action correspondante dans notre jeu s'écoule un temps très court, suffisamment court pour que le joueur ne se rende pas compte d'un quelconque décalage. Arbitrairement on a choisi 1 seconde. Ce temps peut être vu comme inférieur au temps entre deux calculs complet de la part de l'IA, car on est sûr grâce à Unity que les entrées du joueur seront prise en compte à chaque nouvelle frame.

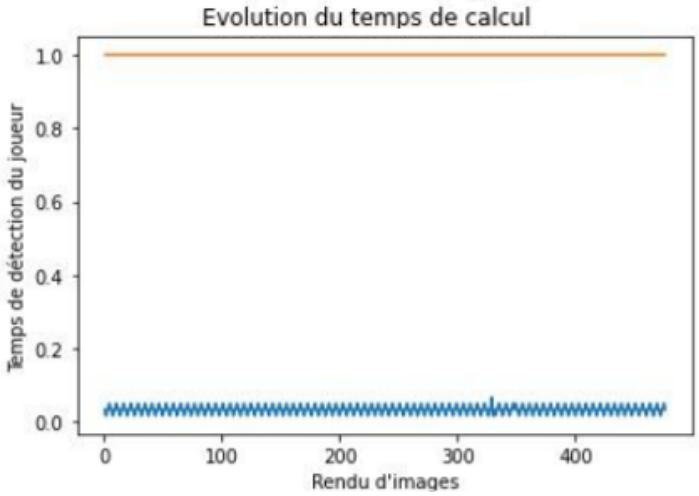


FIGURE 1.11 – Contrainte Algorithme NEAT

Notre contrainte de constance :
On souhaitait que la scène (les images apparaissant à l'écran) s'actualise à intervalle régulier pour assurer une fluidité du gameplay pour le joueur.
On a mesuré, sur une toute séquence de jeu, l'écart temporel entre l'affichage de chaque image (voir image ci-dessous).

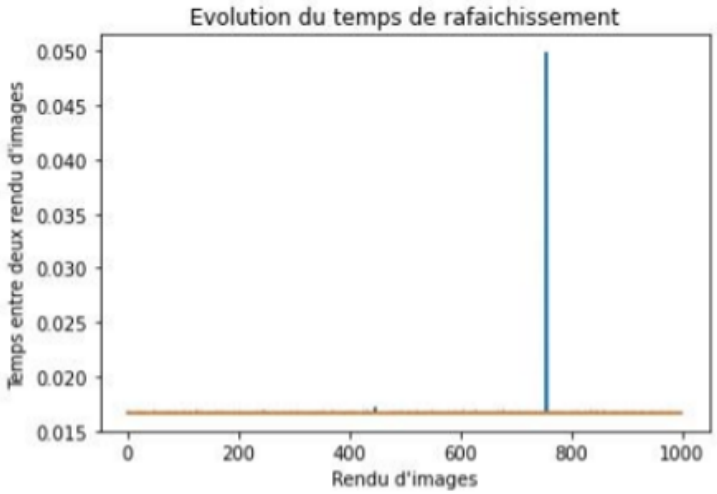


FIGURE 1.12 – Contrainte Algorithme NEAT

On souhaitait avoir 1/60ème de seconde entre chaque image (0.017s). Avec Unity on peut fixer le rafraîchissement, on l'a quand même testé et on est en effet constant à ce 0.017s. Malgré tout, il reste un pic important au niveau de l'itération 780, probablement dû au garbage collector.

Finalement, toutes nos contraintes TR initiales sont satisfaisantes.

Dans le cas de l'algorithme A*, les contraintes semblent globalement très proches de A*, sauf concernant la contrainte d'immédiateté.

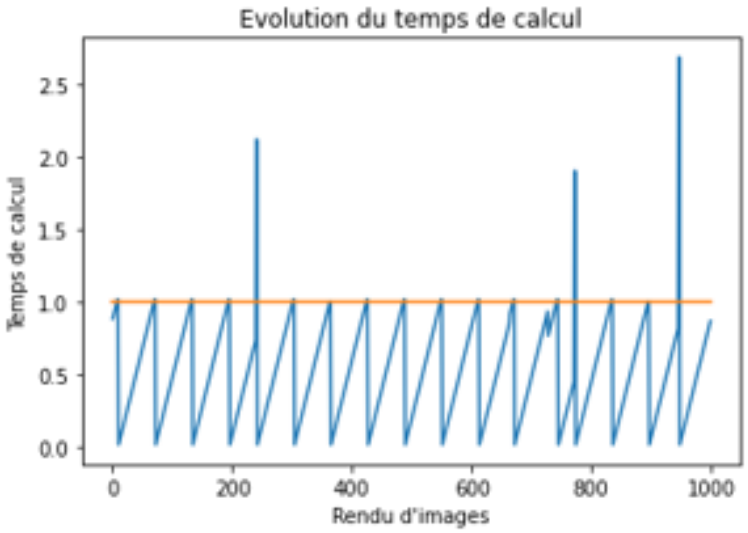


FIGURE 1.13 – Contrainte Algorithme A*

Cette différence s'explique par le fait que NEAT peut être exécuté à chaque nouvelle frame, car le temps de calcul du réseau de neurone est faible. A* en revanche doit être exécuté dans un thread et le temps de calcul étant un peu long, nous avons mis en place un cooldown d'environs 1 s (déterminé par nos essais) pour laisser le temps au parcours de graphe de se finir avant de relancer la recherche.

## 1.3 Pistes d'améliorations

### 1.3.1 Amélioration de A*

Le système de déplacement des ennemis se basant sur deux types de supports différents (NavMesh Surface pour la fonctionnalité de patrouille et Grille de nœuds pour la chasse avec A*), il serait judicieux de retravailler la gestion est déplacements de A* et de la patrouille, pour les rendre compatibles avec l'utilisation du rigidbody (objet soumis à la physique) utilisé pour les autres agents. Nous pourrions alors l'utiliser avec `AgentController` afin de l'implémenter en jeu.

Idéalement nous aurions souhaité utiliser notre propre implémentation de l'algorithme A*, une piste d'amélioration serait donc d'apporter davantage de temps à la mise en place de la grille de nœuds sur l'arène afin de n'utiliser que nos propres implémentations.

## 1.4 Conclusion

Dans ce projet, nous avons mis en place un système de jeu temps réel dans lequel un joueur se mesure à des IAs.
Parmi nos IAs, l'algorithme NEAT est celui qui a permis au mieux de respecter nos contraintes temps réel.
Même si finalement notre IA n'est pas assez performante pour poser un véritable challenger à un joueur humain, nous restons très satisfaits de notre rendu.
Différentes pistes d'améliorations pourraient être d'entraîner une même IA sur les différentes scènes plutôt qu'en faire une par arène, on pourrait aussi bien sûr améliorer les graphismes et textures du jeu.
Malgré tout, ce projet qui était pour nous notre découverte d'Unity nous a permis d'en avoir une première approche assez conséquente et fut très plaisant et enrichissant.