Rapport MOSIMA Projet 2

Abderrahmane CHAABANI - Frederic FERNANDES

January 2017

1 Introduction

Dans ce projet, deux agents s'affrontent dans une arène contenant un terrain à l'altitude irrégulière. Cherchant à se tirer dessus et à éliminer son adversaire, nous avons cherché à doter l'un des agents d'un comportement lui donnant les meilleurs chances de gagner.

2 Mise à niveau et amélioration de la plateforme

Avant de pouvoir commencer le projet, il y avait des améliorations de des changements à faire sur la plateforme "ACTC". Il y avait des bugs et des crashs à résoudre et certaines fonctions de respectées pas leurs buts. Il y a aussi le fait que la plateforme est utilisée pour un projet d'apprentissage, et comme tout algorithmes d'apprentissage il faut réaliser plusieurs parties pour apprendre.

2.1 Résolution des crashs de la plateforme

Une plateforme qui crash, c'est toute l'apprentissage qui est perdu ! Les crashs sont due au fait que **JADE** est multi-threadé. Si un agent (un thread) vient à modifier le monde de **JMonkey**, alors que celui-ci est en train de mettre à jour le monde (fonction update() de la classe SimpleApplication, les threads entrent en conflits et il y a un crash. La solution est donc de faire en sorte que les threads de **JADE** de change pas le monde en même temps que **JMonkey**. La solution que nous avons utilisée est l'utilisation de **Mutex** lorsqu'une fonction ou un bout de code change le monde. Le moteur de **JMonkey** utilise la notion d'arborescence de nœud, un nœud étant un élément dans le monde.

```
rootNode.attachChild(bulletNode);
```

Ici on attache au nœud principal un nouveau nœud *bulletNode*. C'est ce style d'instruction qui modifie le monde et qu'il faut entourer de **Mutex** pour qu'elles puissent être appelées par les threads de **JADE**.
Voici un exemple :

```
synchronized (rootNode) {
    Principal.lockUpdate.lock();
    try {
        rootNode.attachChild(fire);
    } finally {
    Principal.lockUpdate.unlock();
    }
}
```

Nous utilisons un seul **Mutex** *Principal.lockUpdate* , qui est bloquant si un autre thread **JADE** ou **JMonkey** change le monde. Voici la fonction *Update()* qui elle aussi est sécurisée :

```
@Override
        public void update() {
                synchronized (this) {
                        Principal.lockUpdate.lock();
                        try {
                                super.update();
                        } finally {
                                Principal.lockUpdate.unlock();
                        }
                }
        }
```

Il y a aussi une synchronisation au démarrage de **JADE** et **JMonkey**. On attend que le monde soit bien crée (*non plus avec un wait(x)*), mais en utilisant des signaux.
La fonction **main()** de la classe **princ.Principale**.

```
public static void main(String[] args){
    lockUpdate = new ReentrantLock();
    //0) Create the environment
    env = Environment.launchRandom(64);
    synchronized(env){
        try {
                System.out.println("Wait JMonkey ending loading.");
                env.wait();
                System.out.println("Start Jade ");
                emptyPlatform(containerList);
                ...
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

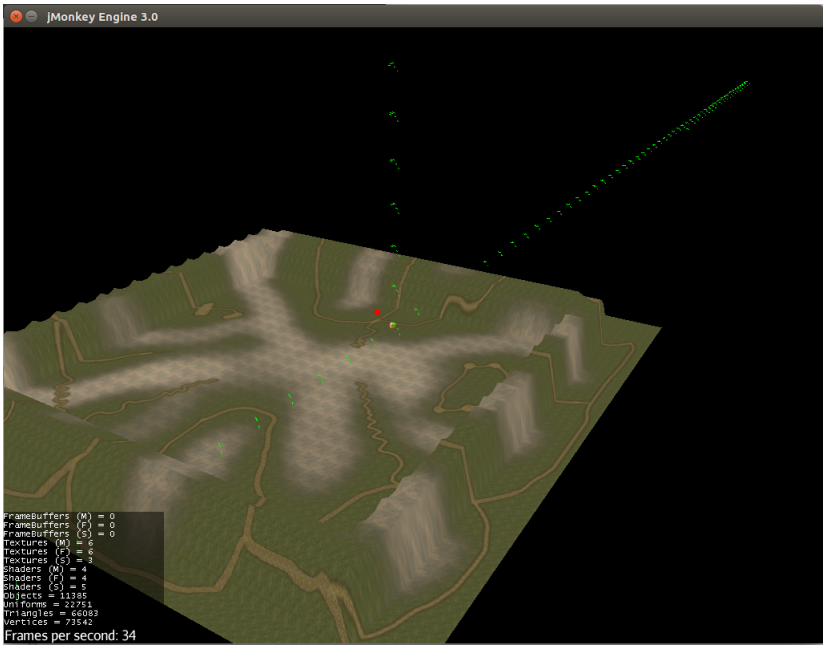
L'instruction *wait()* est bloquant jusqu'à ce que **JMonkey** est fini sont initialisation.
Alors que pour réveiller l'objet en attente on utilise *this.notify()*.
Voici la fin de la fonction *simpleInitApp()* de la classe **env.jme.Environment.java**

```
@Override
        public void simpleInitApp() {
                synchronized(this){
                ...
                        /***
                         * NOW JMonkey is ready for Jade
                         */
                        this.notify();
                }
        }
```

Au final, nous avons réussie à produire une plateforme **Stable** pour commencer les simulations et l'apprentissage.

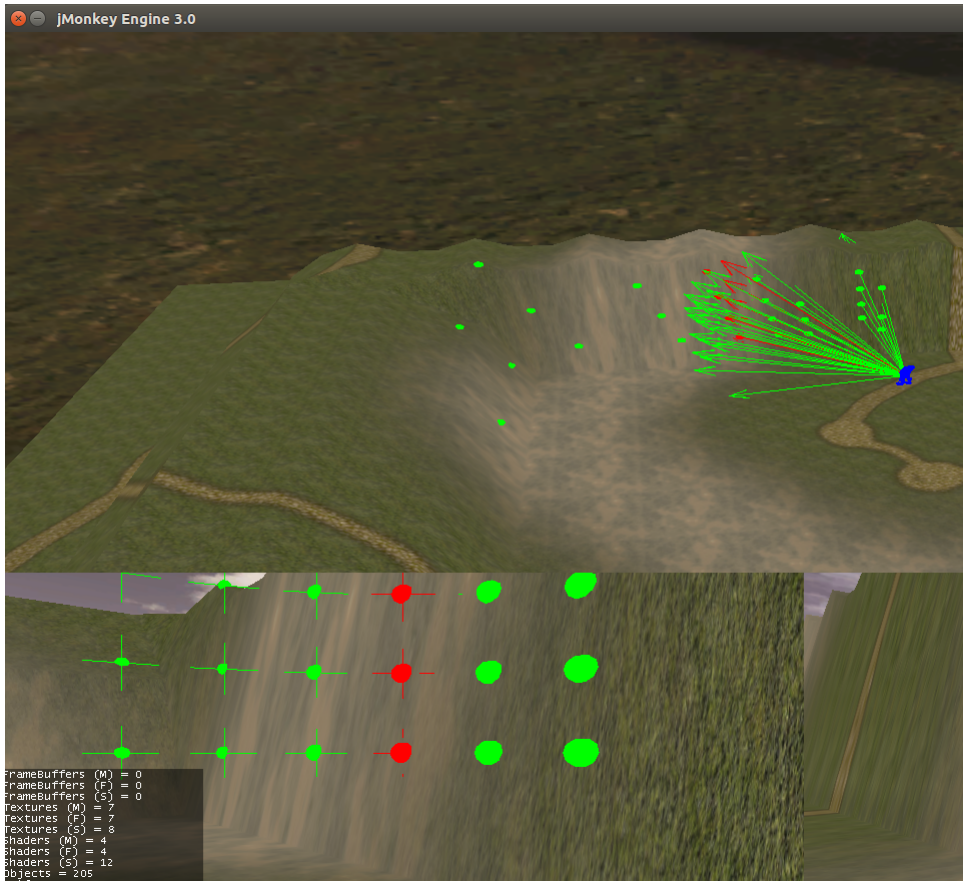
2.2 Modification de certaines fonctions

Certaines fonctions avaient un comportement incorrect, notamment la fonction *observe()*. Nous parlerons dans cette partie uniquement de cette fonction, car c'est la plus importante pour l'apprentissage. Cette fonction a pour but de retourner une **Situation** du monde vu par un Agent à l'instant **t**. On comprend facilement que si cette fonction renvoie une fausse représentation du monde, il est impossible d'avoir un apprentissage qui converge vers nos objectifs.



Visualisation des rayons lancées par la fonction

Comme on peut le voir, les rayons *(trait vert)* sont lancés n'importe comment. Nous l'avons donc modifié afin d'avoir un maillage (**Gros grain**) du champ de vision de l'agent.



Nouvelle visualisation des rayons lancées par notre fonction

Pour le débug de notre fonction *observe()* nous affichons les **RayCasts** par des flèches vertes (*rouge pour les rayons d'angles 0*), les points d'origines et les directions des flèches sont les mêmes que les RayCasts, mais la longueur n'est pas représentative. Nous affichons aussi les points que chaque **RayCast** a atteint. Tous ces points sont ensuite utilisés pour créer une **Situation**. (*Nous n'avons pas fait de modification à ce niveau là*).

2.3 Préparation pour l'apprentissage

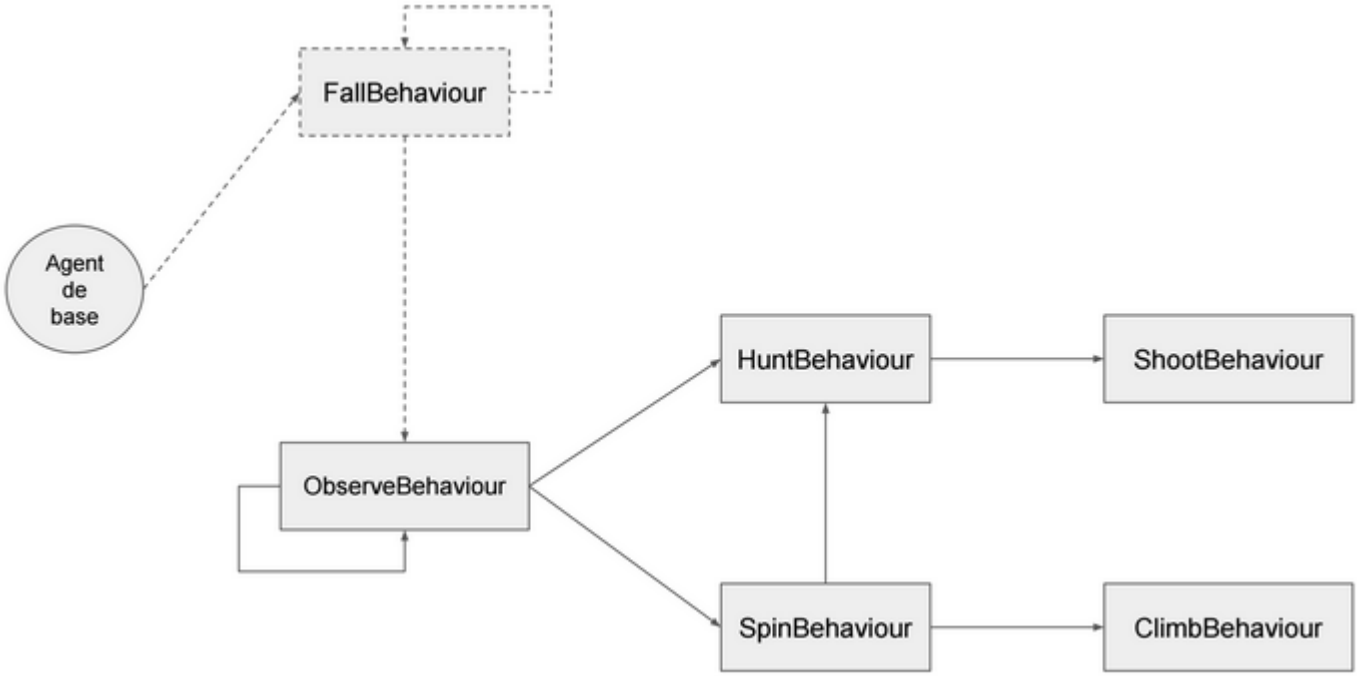
Pour réaliser notre apprentissage nous avons créé un mécanisme permettant de jouer plusieurs parties avec plusieurs cartes. Une partie se fini quand un des deux agents est mort, dans ce cas-là on réinitialise une nouvelle partie. Toutes les **N** parties jouées ou gagnées (*selon les paramètres de l'algorithme d'apprentissage*) une nouvelle carte est générée aléatoirement et est utilisée pour les **N** parties suivantes.
Si une partie est gagnée, la dernière **Situation** est considérée comme *Intéressante*, on là stocke dans une fichier cvs avec les autres **Situation** intéressante dans le but de pouvoir générer une arbre de décision grâce à **Weka**.

2.4 Divers changements

Certaines choses ont était modifiées, comme le mapping des touches de la Fly Camera. L'ajout d'une Sky-box, ou le changement des modèles graphiques des agents.

3 Comportement des agents "intelligents"

Les agents intelligents, de type SmartAgent, disposent de plusieurs Behaviours correspondant à des comportements à adopter selon la situation dans laquelle ils se trouvent. L'organisation de ces behaviours peut être représenté par un arbre comme ceci :



Schematisation du comportement d'un agent

L'agent est initialisé de base avec un SequentialBehaviour, type de comportement permettant d'effectuer dans un ordre donné ses sous-comportements. Il comporte comme premier sous-comportement le FallBehaviour et comme second sous-comportement le ObserveBehaviour.

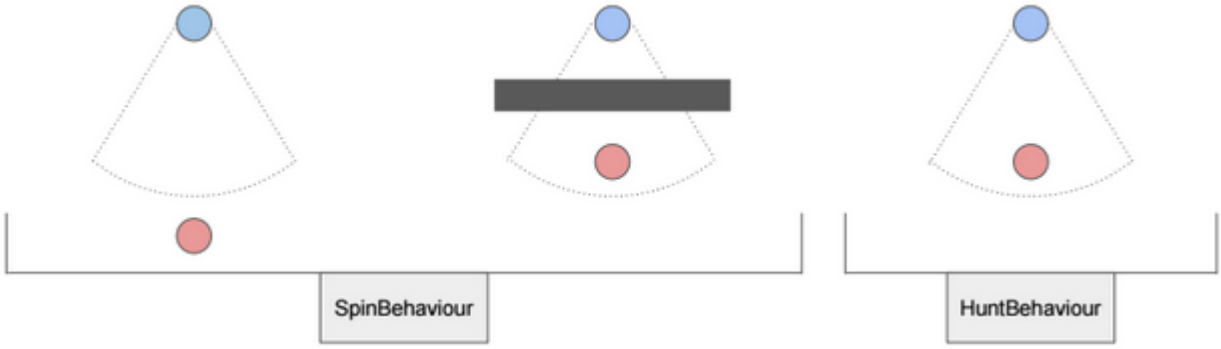
Étudions ces deux comportements "fondamentaux", dont l'exécution est assurée.

3.1 Comportement de chute libre : FallBehaviour

Lors de leurs créations, les agents sont lâchés d'une haute altitude. Ils sont alors en chute libre jusqu'à entrer en contact avec le sol de l'arène. Cependant, on pouvait observer dans certains situations que les agents se tiraient dessus alors qu'ils étaient toujours en l'air, comportement que nous désirions éviter. Le comportement FallBehaviour rempli donc une unique fonction : bloquer les autres comportements des agents jusqu'à ce que ceux-ci atteignent le sol. Pour ce faire, nous avons considéré le fait que les agents sont en chute libre jusqu'à atteindre l'arène. Une fois celle-ci atteinte, sans l'aide d'autres comportements, il n'y a pas de mouvement.
Nous en déduisons donc qu'à deux pas de temps consécutifs (assez espacés pour laisser le temps aux agents de tomber si besoin), si un agent se retrouve à la même position qu'à celle à laquelle il se trouvait au pas de temps précédent, sa chute libre est terminée et il a atteint le terrain. Une fois le sol atteint, le comportement FallBehaviour se termine et n'est plus exécuté par la suite. L'agent passe alors au comportement suivant.

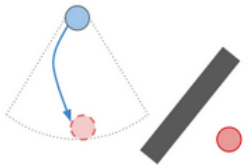
3.2 Comportement de base de l'agent : ObserveBehaviour

Une fois que l'agent à atteint le terrain, il commence alors son exploration, cherchant à éliminer son adversaire de la meilleure manière possible. Il adopte alors un comportement de base qui lui servira alors à prendre des décisions sur le comportement à adopter par la suite : ObserveBehaviour. Ce comportement hérite de TickerBehaviour, et est adopté tout au long de la simulation à intervalle de temps régulier. Lorsque l'agent n'a pas d'informations sur la position de l'adversaire, il commence par observer ce qu'il voit directement et réagit tout d'abord selon s'il voit ou non l'ennemi : voir l'ennemi signifie qu'il est à portée de vue mais aussi qu'il n'y a pas d'obstacle entre eux (comme une surélévation par exemple). Si l'ennemi ne peut-être vu par notre agent, celui-ci va alors analyser ses alentours et adopte alors le comportement SpinBehaviour. Sinon, si l'ennemi est visible, notre agent se met alors en chasse et adopte le comportement HuntBehaviour.



Comportement adopté par l'agent après avoir observé ce qu'il voit directement, lorsqu'il n'a pas d'informations récente sur la position de son adversaire

Cependant, le comportement de l'agent change légèrement lorsqu'il dispose d'informations sur l'ennemi : s'il a aperçu son adversaire récemment mais qu'il l'a perdu de vu, il va chercher à se rendre à la dernière position connue de celui-ci avec un simple moveTo.



Notre agent se souvient de la dernière position de l'agent rouge avant de l'avoir perdu de vue et s'y rend donc

3.3 Comportement de chasse : HuntBehaviour

Ce comportement, ainsi que ceux qui suivent sont des OneshotBehaviours : ils ne sont appelés qu'une seule fois avant d'être détruits. En effet, comme notre agent cherche le meilleur comportement à adopter à chaque pas de temps au sein du ObserveBehaviour, on ne désire pas qu'un comportement dure sur la durée car il pourrait être bénéfique à l'agent à un tick donné, mais détrimentaire au suivant.

L'ennemi est désormais visible par notre agent et celui-ci à gardé en mémoire cette dernière position aperçue puis s'est mis en chasse. Mais comment aborder cette situation ? Notre agent, qui dispose d'une portée de tir différente que sa portée de vue, alors considère deux cas de figure :

- L'ennemi est trop loin pour qu'on puisse lui tirer dessus : Dans cette situation, notre agent se déplace simplement grâce à sa fonction moveTo vers son ennemi.
- L'ennemi est a portée de tir (et le champs de tir n'est pas obstrué) : Notre agent va alors adopter la méthode ShootBehaviour lui servant à tirer sur son adversaire. Il n'y a pas de déplacement de notre agent dans ce cas de figure, car on considère que se rapprocher de l'ennemi s'il est possible de lui tirer dessus n'est pas forcement avantageux. Si nous somme dans une position de force (en amont par exemple) ou dans une position ou l'ennemi ne peut pas nous voir (si par exemple l'angle vers lequel il est possible de regarder vers le bas est plus grand que l'angle vers lequel il est possible de regarder vers le haut), notre agent n'a pas intérêt à bouger.

Analysons donc maintenant le comportement de tir de notre agent

3.4 Comportement de tir : ShootBehaviour

Le comportement de tir d'un agent est simple : lorsqu'il est appelé, l'agent appelle sa fonction shoot sur l'ennemi. Nous avons créé un comportement à part entière pour cette fonction pour éventuellement par la suite ajouter une cadence de tir ou d'autres attributs plus complexe.

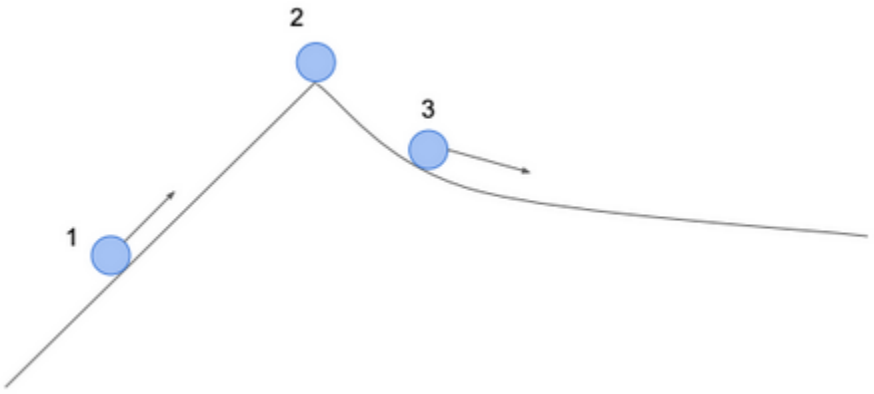
3.5 Comportement d'analyse des alentours : SpinBehaviour

Étudions maintenant le comportement de l'agent s'il n'a pas son ennemi en ligne de vue et qu'il ne dispose pas d'informations récente sur sa position (une information sur la position de l'adversaire est considérée comme récente tant qu'on ne s'est pas encore rendu à cette dernière position sans avoir vu l'ennemi entre deux).

Notre agent se met à tourner sur lui-même : il effectue une rotation vers chacun des points cardinaux, et observe son environnement à chaque rotation. Si lors d'une rotation notre agent aperçoit son adversaire, il adopte alors le comportement de Chasse, décrit plus haut.
Sinon, il doit alors choisir un endroit où se diriger, lui offrant un avantage stratégique. Nous avons déterminé qu'un de ces endroits stratégiques est le point culminant de la carte : celui d'altitude la plus élevée, permettant un champs de vision élevé. Cependant, le champs de vision de notre agent n'était pas illimité, celui-ci stock le point d'altitude le plus élevé qu'il a pu apercevoir lors de ses rotations dans une variable highestPos : ce sera un point d'altitude locale maximum, mais pas forcément globale. Une fois ce point d'altitude locale maximum détectée, notre agent entame alors son dernier comportement : celui d'ascension.

3.6 Comportement d'ascension : ClimbBehaviour

Lors de l'ascension de l'agent afin d'atteindre le point culminant de la carte, notre agent peut se retrouver dans trois situations que nous pouvons schématiser ainsi :



Représentation de profil des différentes situations dans lesquelles l'agent peut se retrouver lors de son ascension

- La situation [1] représente l'ascension de l'agent au point culminant qu'il a pu détecter. Cette opération est effectuée à l'aide de la fonction moveTo de l'agent, celui-ci se dirige directement vers sa cible.
- La situation [2] représente l'agent lorsqu'il est arrivé au point d'altitude le plus élevé qu'il a pu repérer. Une fois au sommet, celui-ci va alors se tourner dans une direction aléatoire et observer son environnement, dans l'espoir d'apercevoir l'agent adverse au tick suivant. Cependant, si ce dernier décide de se cacher ou est coincé dans une position où il ne peut pas bouger, rester au point culminant ne débloquera pas la situation. Notre agent restera donc au point culminant pendant une durée de 5 secondes. Une fois ces 5 secondes passées, notre agent effectue alors une brève patrouille représentée dans la situation [3]
- La situation [3] représente comme décrit ci-dessus la patrouille de l'agent. Cette patrouille sert non seulement à repérer un éventuel agent qui se cacherait ou resterait dans une certaine zone, mais aussi à découvrir par exemple une nouvelle situation intéressante où se placer comme un nouveau point culminant. Une fois qu'il a décidé d'entamer une patrouille, l'agent

se dirige dans une direction aléatoire et se dirige dans cette direction pendant 20 ticks, avant de recommencer à chercher une ascension au point culminant (si bien sur aucun des autre comportements n'a pris le dessus depuis sur ClimbBehaviour).

Tous ces comportements que nous avons décrit jusqu'à maintenant constitue l'ensemble des comportements de notre agent.

4 Résultats expérimentaux

Nous avons testé notre agent de type SmartAgent, constitué des différents comportements que nous avons décrit précédemment, contre un agent ChaseAgent avec un comportement très simple : Si celui-ci voit un ennemi, il s'en approche tout le temps, et lui tire dessus si possible. Sinon, il bouge de façon aléatoire.

Sur une vingtaine de simulations lancées, nous avons observé treize victoires de notre agent qui a réussi à tuer l'adversaire en survivant, quatre défaites où notre agent s'est fait tuer et l'adversaire a survécu, et trois matchs nuls où les deux agents se sont entre-tués.

En modifiant les paramètres des agents pour que leurs portées de tir soient égales à celles de leurs portées de vue, le taux de victoire de notre agent se voyait nettement augmenté : seize victoires contre quatre défaites. En effet, en cherchant à atteindre le point culminant, le SmartAgent réussissait à apercevoir son adversaire le premier. S'il pouvait lui tirer dessus instantanément, il arrivait généralement à prendre le dessus sur son adversaire. Cependant, si un déplacement était nécessaire avant de pouvoir tirer, le ChaseAgent avait un bon nombre de fois le temps d'apercevoir le SmartAgent venir, et par conséquent les deux se fonçaient dessus jusqu'à pouvoir se tirer dessus, en même temps. En tirant en même temps, les deux agents tiraient parfois avant de mourir ce qui entraînait les matchs nuls. Par ailleurs, les probabilités de toucher étant inférieur à 1, parfois un agent qui se retrouvait à un certain désavantage réussissait, par chance à remporter le duel.

5 Améliorations

Ayant rencontré un bon nombre de problèmes dus à swipl, nous n'avons pas réussi à intégrer de façon utile Prolog à notre projet. Cependant, les états que nous pourrions représenter en Prolog correspondent aux différents états entraînant l'activation des Behaviours appropriés. La transition de Java à Prolog de certaines fonctionnalités est donc facilitée.
Par ailleurs, ayant modifié les fonctions observe et intersects (en observe2 et intersects2) afin de pouvoir détecter les points entourant l'agent, nous sommes en mesure de mieux représenter une Situation que grâce à l'altitude moyenne, max et min de ce qu'un agent voit. Nous pourrions donc représenter des situations complexes (au font d'un gouffre, dos à un mur, dans un couloir) afin d'effectuer un apprentissage avancé en pouvant détecter des situations jugées interessantes plus complexes que ce dont disposait initialement.