



# Projet Tutoré - Rapport de mi-projet

Conception et développement d'une simulation de capture  
de drapeau et mise en place d'agent autonome

HIRTZ - JEAN-BAPTISTE - JUNG - LE NALINEC - NAIGEON  
RA-IL2

Tuteur : Amine BOUMAZA  
2024/2025

<b>Introduction.....</b>	<b>3</b>
Installation.....	3
<b>Analyse.....</b>	<b>4</b>
Découpage fonctionnel du projet.....	4
Modèles UML utilisés.....	5
MVC.....	5
Multi Model.....	5
Evolution par rapport à l'étude préalable.....	7
Différence sur le planning.....	7
<b>Réalisation.....</b>	<b>7</b>
Réalisations logicielle.....	7
Réalisations d'intelligence artificielle.....	10
Structure d'un agent.....	10
Entrées (perceptions).....	10
Sorties (actions).....	11
Arbre de comportement.....	11
Apprentissage.....	11
ECJ.....	12
Fonction d'évaluation.....	12
Tests de validation.....	13
Difficultés rencontrées.....	13
Application.....	13
ECJ.....	14
Bugs et "Edge-Case".....	14
<b>Planning de déroulement du projet.....</b>	<b>15</b>
<b>Présentation d'un élément dont nous sommes fier.....</b>	<b>17</b>
Damien – L'affichage de perceptions et l'entraînement.....	17
Adrien – Raycast et moteur.....	17
Grégoire – MVC.....	17
Tibère – Editeur de carte.....	18
Evan – Développement des agents d'arbre de Décision :.....	18
<b>Planning pour les trois itérations restantes.....</b>	<b>19</b>
<b>Conclusion.....</b>	<b>20</b>



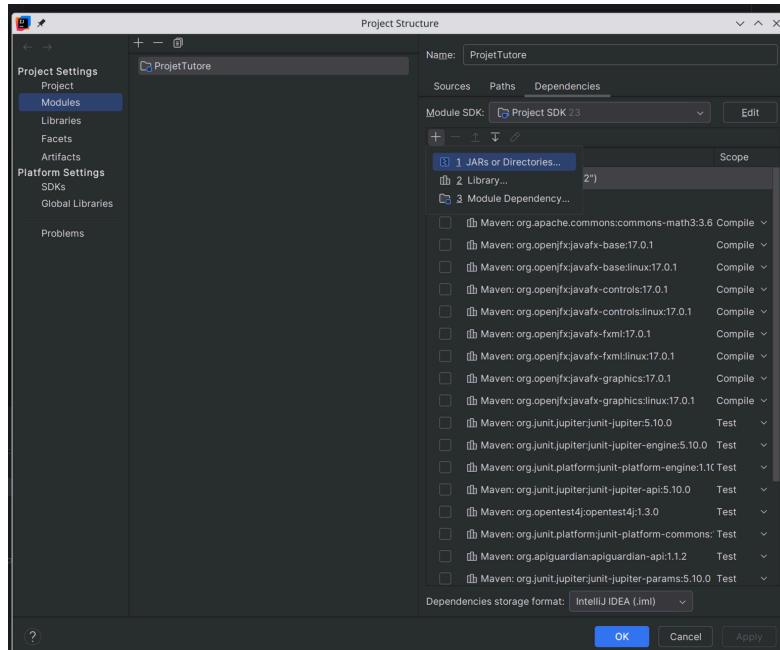
# Introduction

Notre projet tutoré a pour objectif de concevoir et développer une simulation du jeu « Capture The Flag » en Java. Ce jeu, qui se joue généralement en extérieur et oppose deux équipes, consiste à s'introduire dans le camp adverse pour s'emparer de son drapeau et le ramener dans son propre territoire, tout en défendant son propre drapeau. Ce principe allie stratégie offensive et défensive, ainsi qu'une coordination étroite entre les membres de l'équipe. Notre but est donc à terme d'utiliser ce support développé par nos soins afin d'expérimenter diverses méthodes d'intelligence artificielle pour d'optimiser la performance des agents impliqués dans le jeu. Ce document a pour objectif, après quatre itérations réparties sur un semestre, de faire le point sur les tâches accomplies durant cette période, de documenter nos avancées, les problèmes rencontrés ainsi que les solutions mises en œuvre, et enfin, de présenter notre point de vue actualisé sur le projet ainsi que nos objectifs futurs.

## Installation

Pour lancer le projet, il suffit de charger les dépendances grâce au fichier **pom.xml** de Maven et d'ajouter manuellement la bibliothèque **ECJ**, disponible dans le dossier **/ecj-jar/**.

Avec IntelliJ, l'importation des dépendances Maven est normalement automatique (ou du moins très simple). En revanche, l'ajout manuel de **ECJ** nécessite de passer par un menu spécifique.

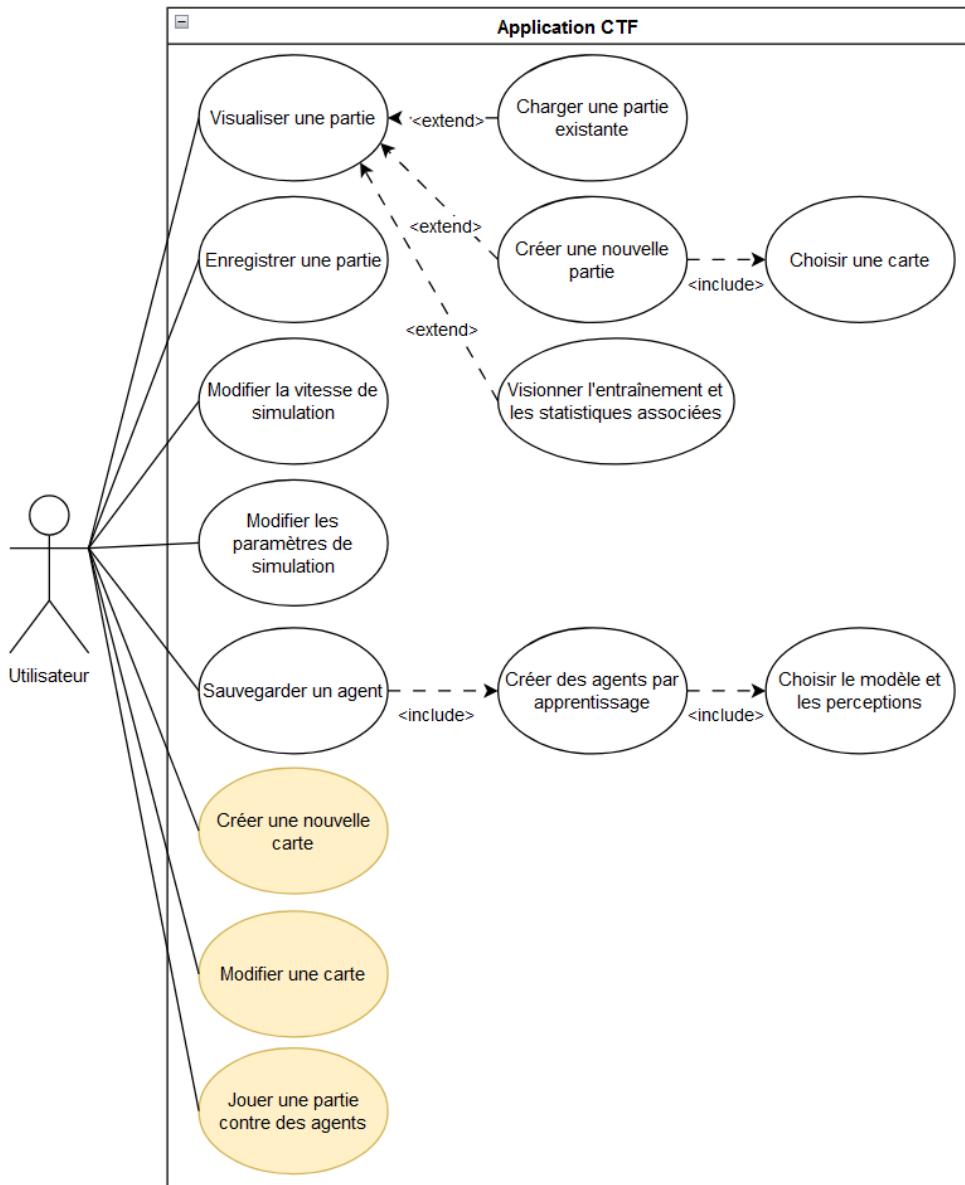


# Analyse

## Découpage fonctionnel du projet

Concernant le découpage du projet en cas d'utilisation, nous ne sommes que très peu éloignés des cas d'utilisation prévus pour l'étude préalable.

Voici ci-dessous le diagramme de cas d'utilisation de l'étude préalable.

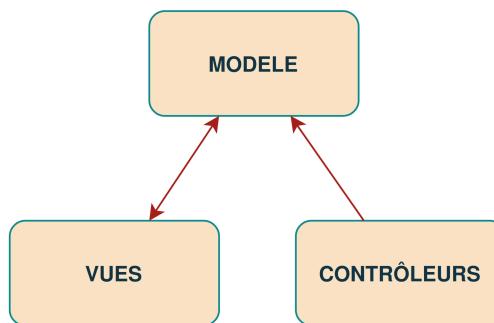


Tous ces cas d'utilisations ont été effectués sauf le cas "Jouer une partie contre les agents" (qui était prévu en plus). Mais nous réfléchissons à la possibilité de rajouter cette fonctionnalité dans les itérations suivantes si nous avons le temps.

## Modèles UML utilisés

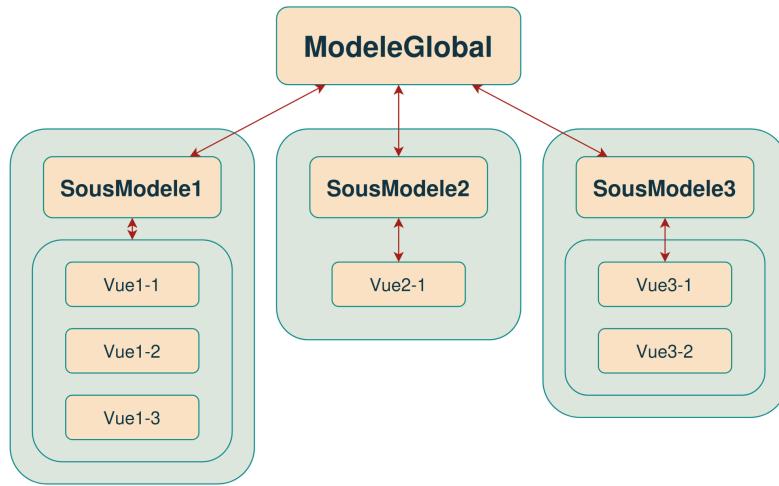
### MVC

Notre système d'affichage est un élément central du développement de notre application. Pour ce faire, nous avons décidé naturellement d'utiliser le modèle MVC. Durant la 1ère et 2ème itération, nous avons utilisé le MVC sous sa forme classique. C'est-à-dire, un modèle qui stocke et gère l'ensemble d'information de l'application, des vues qui s'actualisent par rapport au donnée du modèle, et des contrôleur pour modifier les données par rapport aux actions des utilisateurs.

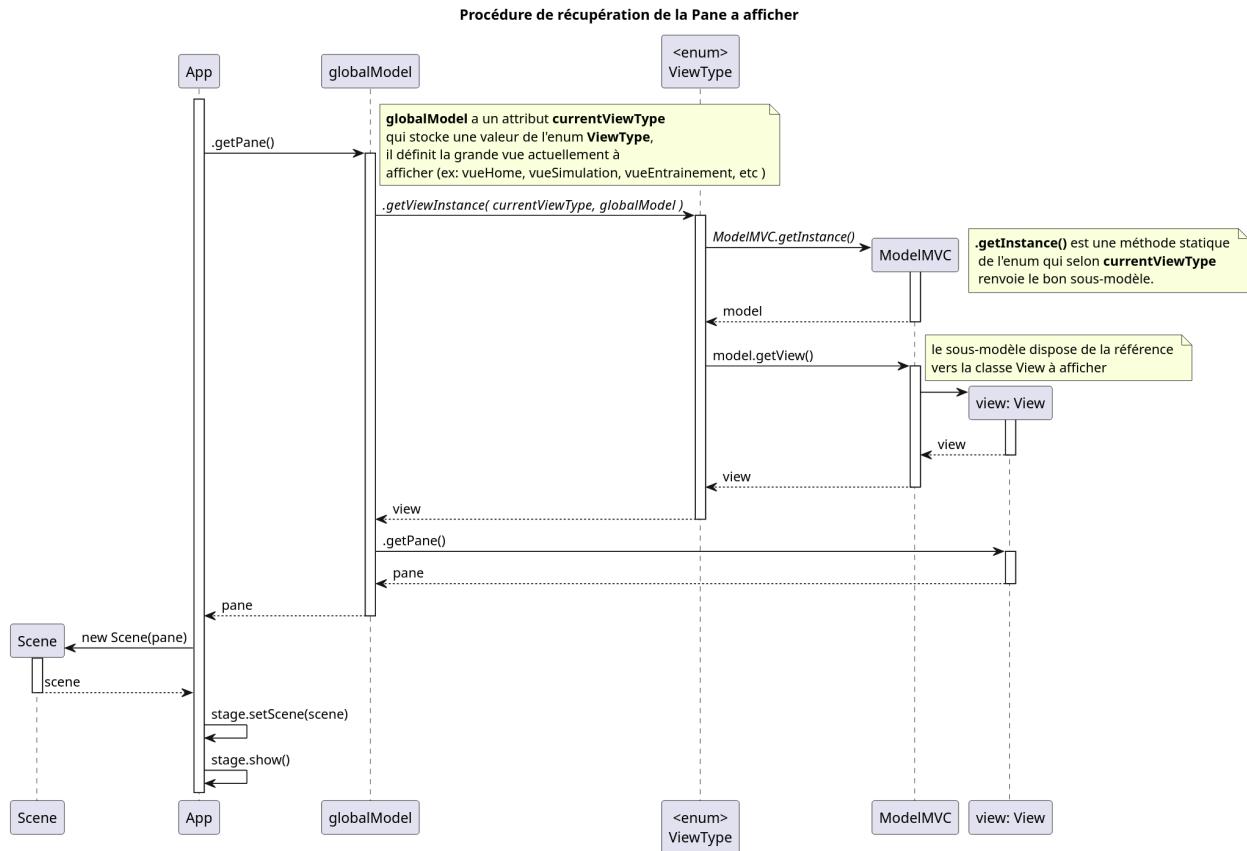


Mais rapidement plusieurs problèmes nous ont dérangés. Tout d'abord, le modèle unique pour de petites applications ne pose pas trop de problèmes, mais nous avons plusieurs grandes vues distinctes, avec des nombreuses sous-vues, ce qui rend rapidement le modèle très fourni qui rend son utilisation fastidieuse. De plus, JavaFX n'est pas une très bonne solution pour faire quelque chose d'esthétique facilement. Pour répondre à ces problèmes, nous avons décidé de faire une refonte complète du MVC.

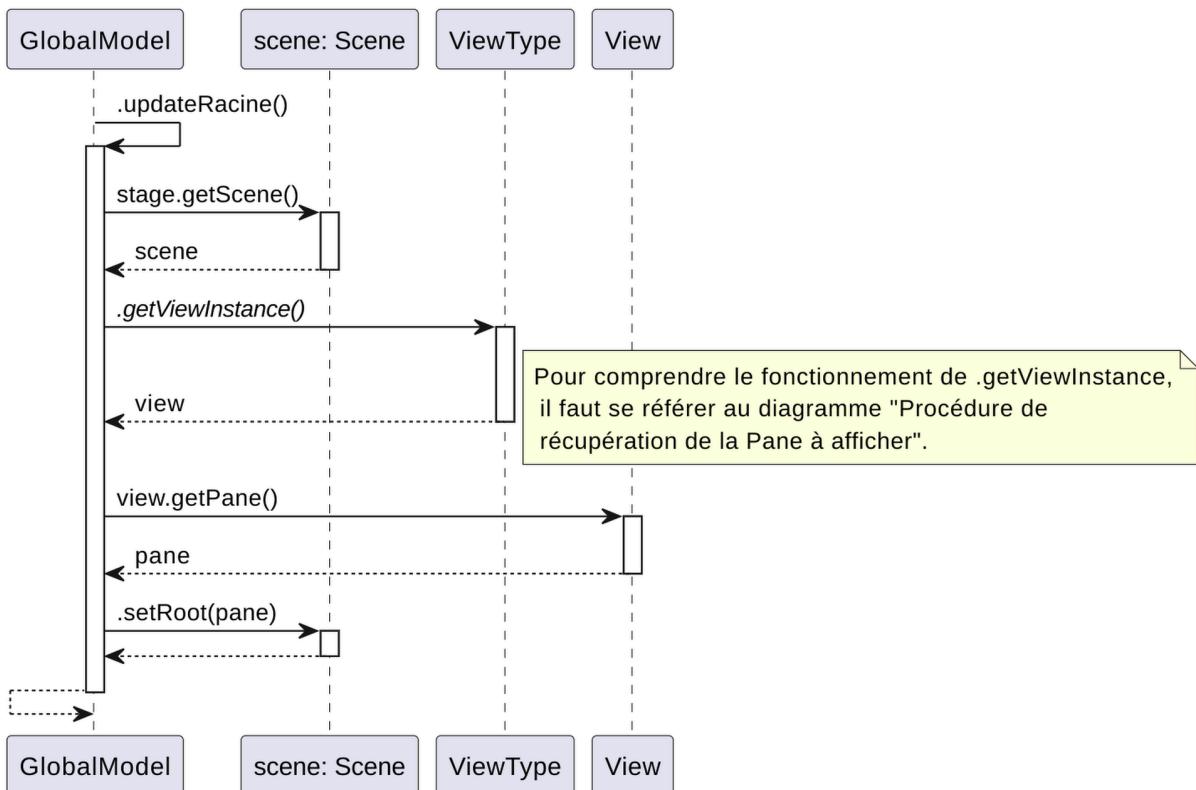
L'idée est simple, comme notre application va avoir plusieurs grande partie (simulation, éditeur de carte et entraînement), on va mettre en place des sous-modèles dédiés à chaque partie. Il faut voir cela comme ci-dessous :



Pour entrer dans le détail, l'application dispose d'un modèle global qui regroupe uniquement les informations utiles à l'ensemble des grandes parties. Pour comprendre son fonctionnement, on peut regarder la description de la procédure d'affichage d'une vue.



### Procédure de mise à jour de la vue



## Evolution par rapport à l'étude préalable

Par rapport à l'étude préalable, nous n'avons pas beaucoup de modifications importantes. Nous avons gardé les fondements globaux que nous avions imaginé.  
Cependant il y a tout de même quelques différences notables dont nous parlons ci-dessous.

### Différence sur le planning

Par rapport au planning nous avons mis plus de temps que prévu pour développer les fonctionnalités de l'application. Vous retrouverez dans la partie "Planning de déroulement du projet" un tableau récapitulatif de nos attentes et de ce que nous avons fait par rapport à chaque itération avec les ajouts. Le tableau présente aussi qui a fait quoi.

# Réalisation

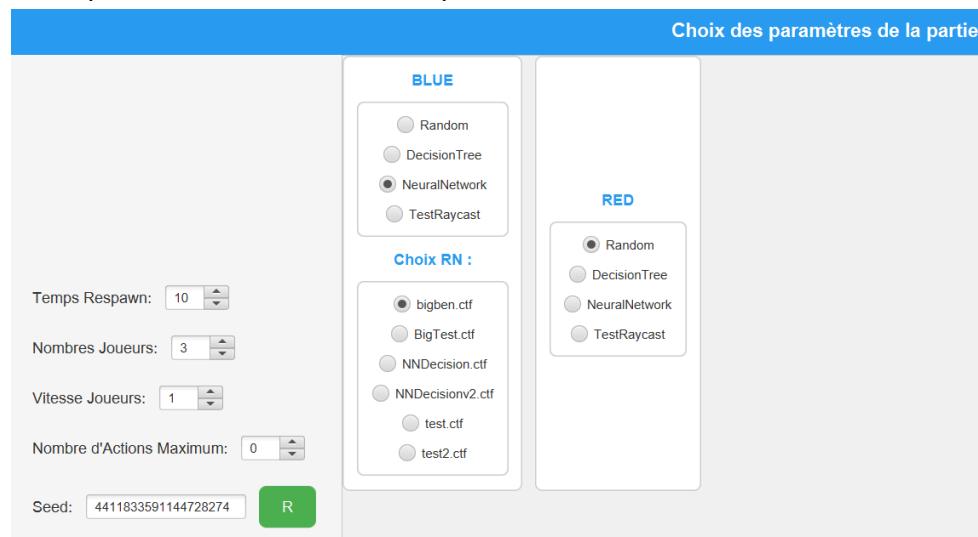
## Réalisations logicielle

Nous avons présenté l'architecture logiciel dans la partie "Modèles UML utilisés". Voici ci-dessous quelques captures d'écran de notre application pour illustrer les fonctionnalités développées. Certaines captures ont été coupées pour qu'on puisse lire le texte.

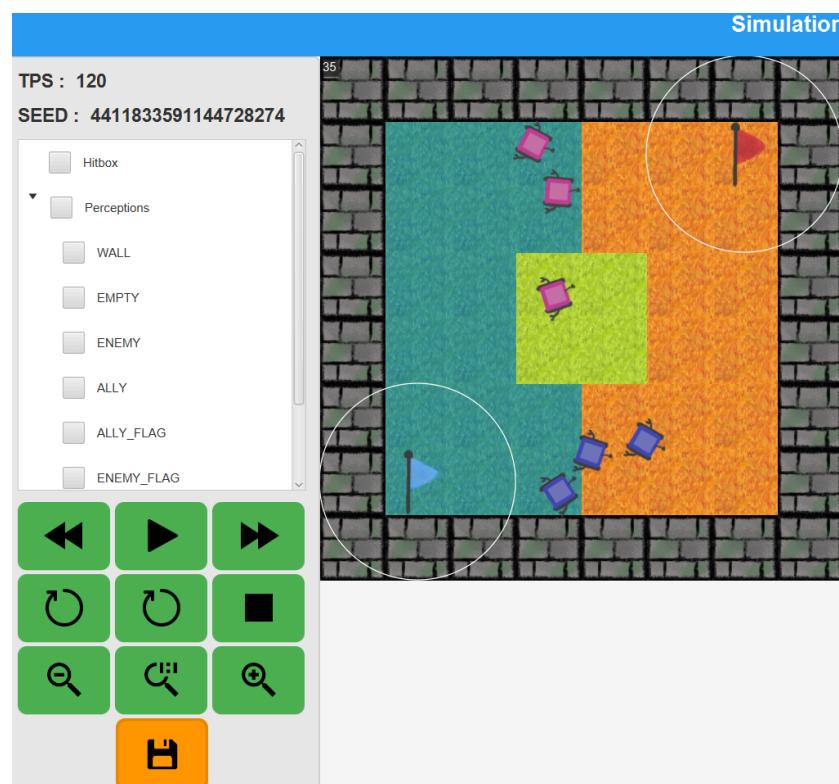
Menu général :



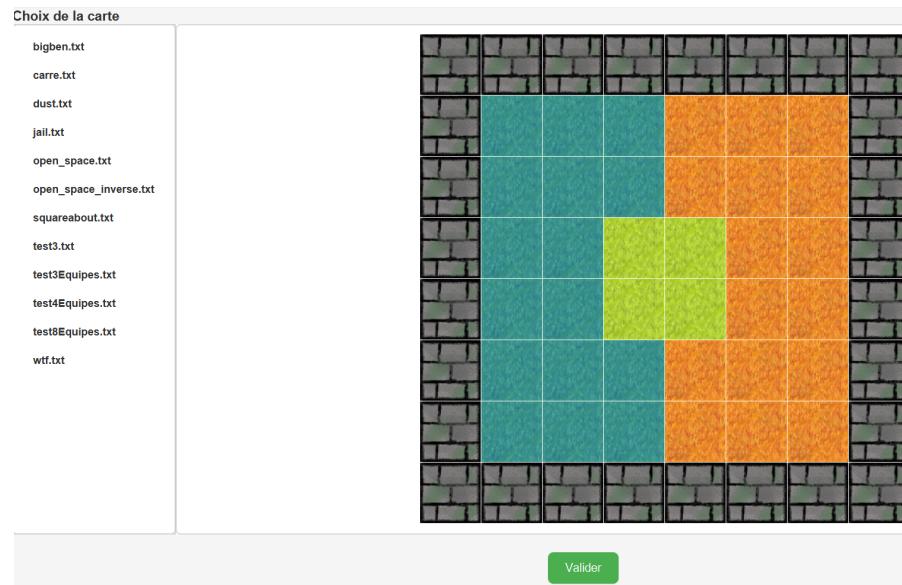
Partie choix des paramètres d'une nouvelle partie :



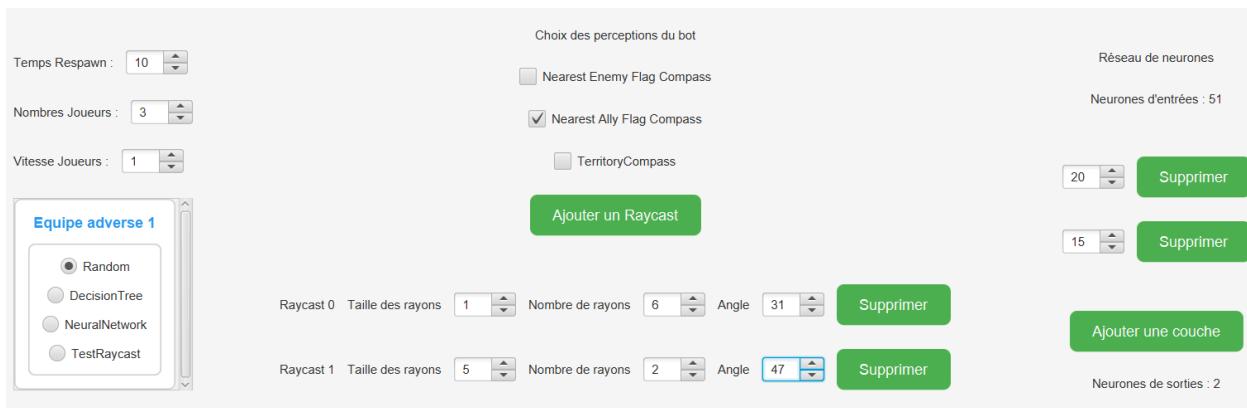
Menu Simulation avec le choix des perceptions et les différents paramètres de visualisation de partie :



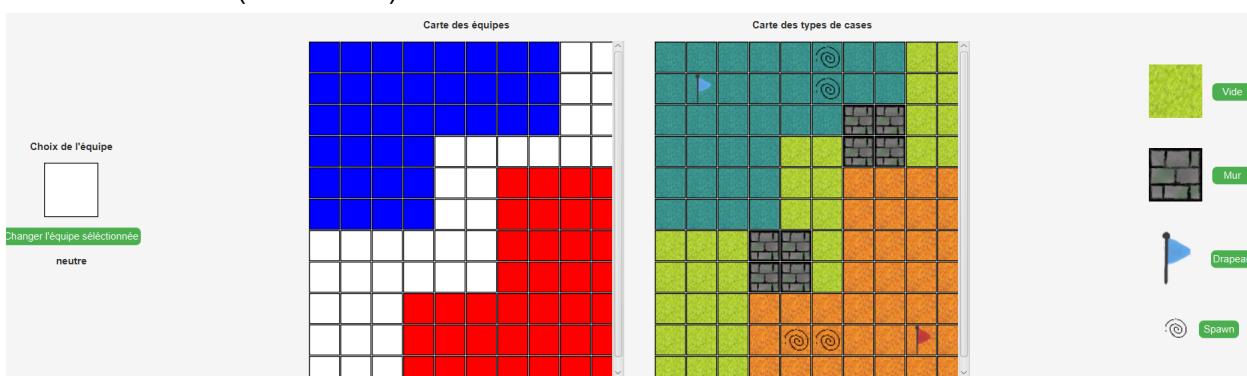
Menu choix d'une carte (présent dans la création d'une nouvelle partie et dans le menu apprentissage) :



Menu de sélection des paramètres pour l'apprentissage :



Menu de création (et d'édition) d'une carte :



## Réalisations d'intelligence artificielle

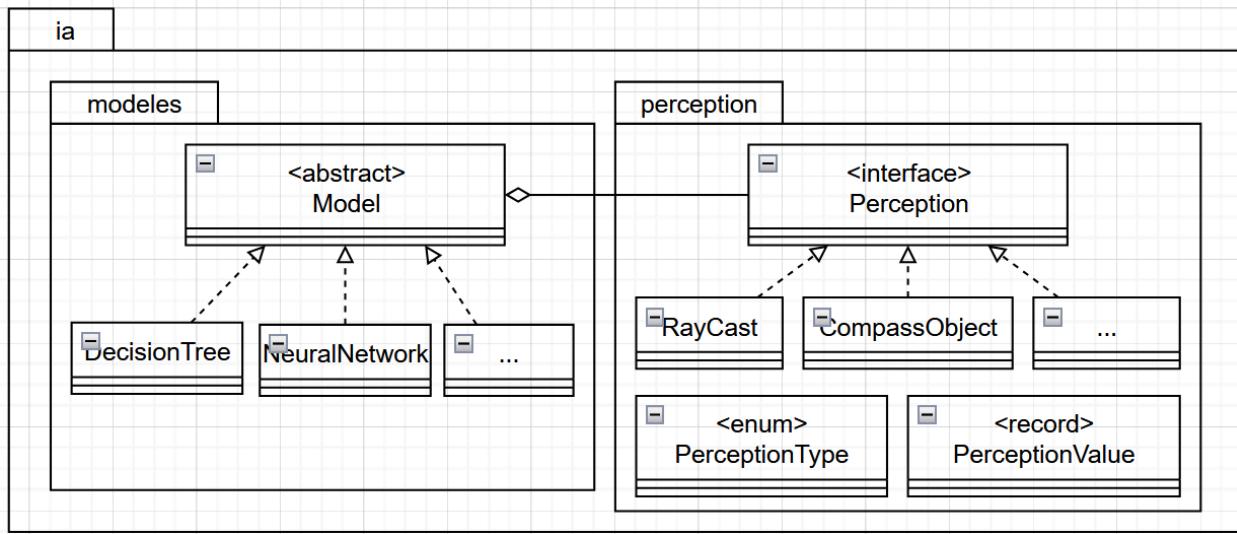
Concernant la partie IA, nous avons travaillé sur deux méthodes pour jouer : l'arbre de comportement et les réseaux de neurones. Créer l'arbre de comportement à été la première partie du projet, une fois que ce dernier était fonctionnel, nous avons développé la partie réseau de neurone et apprentissage. A l'issue de l'itération 4, l'arbre de comportement que nous avons développé est très efficace et utilise même des rôles aléatoires. Nous avons un code pour les réseaux de neurones qui fonctionne bien et qui est capable d'apprendre, cependant il nous manque l'export automatique du réseau de neurone et l'affichage de la fitness dans l'interface.

### Structure d'un agent

Nous avons structuré les agents comme des robots : ils perçoivent l'environnement via des capteurs et effectuent des actions en fonction de ces visions. Ainsi le but de l'arbre de comportement et des réseaux de neurones est de prendre les perceptions et les transformer en action, si possible en jouant le mieux possible.

#### Entrées (perceptions)

Les agents perçoivent le monde grâce à leurs perceptions, une liste de fonction qui prend en entrée le monde et retourne les informations que l'agent aura le droit d'utiliser pour ce diriger : les agents ne sont pas omniscients. Nous avons deux types de perceptions : les compas et les rayons. Les compas donnent la direction et la distance vers des éléments spécifiques (drapeau le plus proche, ennemi le plus proche, mur le plus proche..) tandis que les rayons sont quant à eux des yeux qui permettent à l'agent de voir son environnement proche, les rayons sont décrits selon 6 valeurs, les 3 premières représentent le point de contact, et les 3 autres servent à identifier la nature de l'objet touchée (type, équipe..).



*Package “ia” du projet capture the flag,  
le “modèle” représente l’ia et possède différentes perceptions*

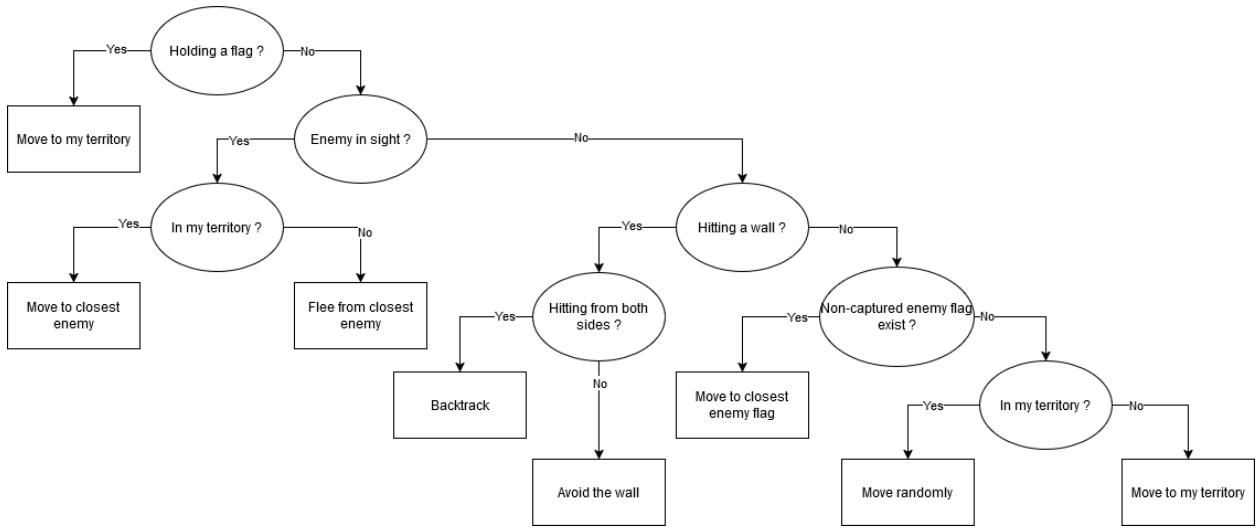
### Sorties (actions)

A chaque tour de simulation, les agents vont percevoir le monde à l'aide de leur perceptions, ensuite l'agent (un arbre de comportement ou un réseau de neurone) va transformer ces perceptions en deux sorties :

- La vitesse : valeur entre 1 et -1 représentant la vitesse à laquelle l'agent veut se déplacer
- La direction : valeur entre 1 et -1 représentant la direction dans laquelle l'agent veut tourner (-1 à gauche, 1 tout droit)

### Arbre de comportement

Le premier agent que nous avons développé (après l'agent aléatoire) est un agent fonctionnant à l'aide d'un arbre de comportement. Ce dernier peut être décrit assez simplement : Il se déplace vers le drapeau ennemi le plus proche en esquivant les ennemis et les murs sur le chemin, puis rapporte le drapeau dans son territoire. L'agent reçoit aléatoirement un rôle (attaquant ou défenseur), s'il est attaquant, il utilisera l'arbre décrit, dans le cas contraire il patrouille autour du drapeau allié le plus proche en attaquant les ennemis sur son passage.



*Diagramme représentant l'arbre de comportement d'un attaquant*

## Apprentissage

La seconde méthode que nous utilisons pour convertir les perceptions en action est un réseau de neurones. Ce réseau prend en entrée les valeurs numériques des perceptions et ressort directement l'action qu'il souhaite prendre.

Pour entraîner le réseau, nous n'utilisons pas de données existantes, à la place nous expérimentons différents réseaux en les faisant jouer plusieurs parties, évaluons les performances des différents réseaux avec l'aide d'une fonction d'évaluation, puis utilisons ces expériences pour modifier le réseau.

## ECJ

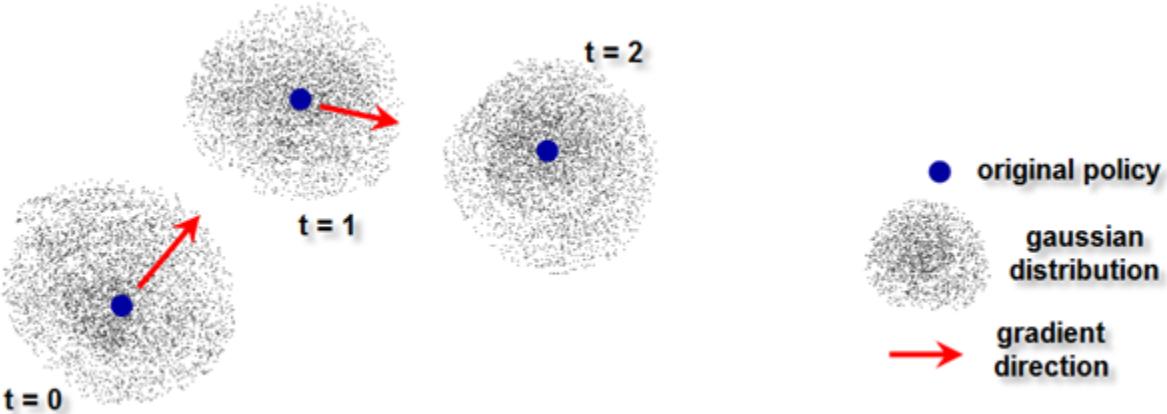
ECJ est une bibliothèque permettant de faire de l'évolution génétique pour maximiser ou minimiser des fonctions à l'aide de populations :

Tout d'abord, un réseau va être généré (point bleu sur l'image), ensuite une population d'enfants va être créée autour de ce réseau (points noirs dans l'image). Tous ces enfants vont être évalué

Tout d'abord la génération d'une population de vecteur double selon les règles qui lui ont été données (valeurs min et max par exemple). La seconde étape consiste en l'évaluation de chaque individu de la population pour obtenir sa note, c'est cette note qui sera utilisée dans la troisième étape. On finit par ne conserver que les individus avec la meilleure note pour créer un individu qui sera un croisement des meilleurs individus qui ont été retenus.

Actuellement nous utilisons l'algorithme : Covariance Adaptation Evolution Strategy.

CMA-ES utilise un seul individu dans sa population, c'est seulement lors de la production de la génération suivante qu'il va créer n autres individus, les évaluer et créer le prochain individu à partir des meilleurs.



### Fonction d'évaluation

Actuellement notre fonction d'évaluation se présente de cette manière :  
 $\text{scoreEquipeAlliee} * \text{poidsAlliee} - \text{scoreEquipeEnnemi} * \text{poidsEnnemi}$ .

Avec `poidsAlliee` et `poidsEnnemis` des valeurs permettant de moduler l'importance des scores de chaque équipe.

Le score d'une équipe est défini comme suit :

`scoreEquipe = 10000 - recordDistanceAgent - recordDistanceDrapeau`

où `recordDistanceAgent` est la distance minimale à laquelle les agents ont réussi à s'approcher des drapeaux (vaut 0 quand un agent a capturé le drapeau)

et `recordDistanceDrapeau` est la distance minimale à laquelle les agents ont réussi à approcher les drapeaux de leur territoire. (vaut 0 quand un agent a ramené le drapeau à sa base)

Ces deux variables tendent vers 0 à mesure que les agents s'approchent des drapeaux, c'est pourquoi nous soustrayons cette valeur à une valeur arbitraire (ici 100000), afin que le score croît au lieu de diminuer. Cette croissance permet à la fonction finale d'incorporer facilement le score allié et ennemis.

## Tests de validation

Lors de la première itération, nous avons créé plusieurs classes de test sur l'agent aléatoire et la carte.

Pour les autres itérations, nous n'avons pas continué à développer des tests. Nous sommes partis du principe que notre application est très visuelle (utilisation de javaFX avec les boutons, les champs etc...) et qu'il est donc plus simple d'effectuer des tests lorsque nous développons les fonctionnalités pour vérifier qu'elles fonctionnent.

Le choix de créer des classes de tests ne nous a pas paru très pertinent.

## Difficultés rencontrées

### Application

Nous avons eu des difficultés sur la structure de l'application avec le modèle MVC. La première version de notre application était simpliste (un seul modèle contenant toutes les informations nécessaires pour naviguer entre les menus), des menus reliés entre eux simplement et du javaFX créé totalement avec java.

Nous sommes donc passé à un modèle plus complexe mais mieux structuré avec plus de possibilités d'évolution (ajout de fxml pour le javaFX fixe, css pour le style et un modèle par grande partie de l'application).

Ensuite, nous avions et avons encore des difficultés sur l'affichage de l'application avec javaFX. La responsivité est compliquée à gérer et l'affichage et la taille des différents composants aussi.

### ECJ

Nous avons eu des difficultés lors de l'installation d'ECJ (pas de tuto simple), lors de l'apprentissage de son fonctionnement (pas d'informations claires, une doc vieille et pas à jour). Mais une fois installé et compris, nous avons pu nous en servir pour l'apprentissage de réseaux.

La première étape a été de créer des réseaux très petit ne disposant que de quelques entrées, les premiers modèles étaient donc composé d'une perception vers la boussole du drapeau ennemis le plus proche, ce premier test nous a permis de voir que ECJ était capable de faire apprendre un réseau pour le faire suivre une boussole.

Nous avons ensuite entrepris de faire apprendre à des réseaux plus grands et plus complexes. Ces premières expérimentations nous ont permis de découvrir que nous utilisions la mauvaise fonction de transfert pour nos réseaux de neurones.

En effet, jusque là nous utilisions la Sigmoïde, cependant cette fonction ne sort des résultats qu'entre 0 et 1, or les agents peuvent fournir des valeurs entre -1 et 1, nous avons pu entraîner un petit réseau qui avait pour perception uniquement 2 rayons de taille 1, capable de reconnaître un mur, mais incapable de contourner par la gauche...

Après avoir changé la fonction de transfert pour n'utiliser que la tangente hyperbolique (ou tanh) nous avons pu constater de meilleurs résultats.

### Bugs et “Edge-Case”

Une grande partie des problèmes rencontrés viennent de cas très précis que l'on ne remarque qu'en itérant un grand nombre de fois des parties aléatoires. Tel que un agent dans un cas très précis qui peut se retrouver entre 4 murs ou un agent qui envoie des drapeaux hors zone.

Cela nous pousse à raffiner de plus en plus les fonctionnalités déjà existantes et d'envisager aux mêmes moments de cas qui n'auraient pas été pris en compte plus tôt.

# Planning de déroulement du projet

	Etude préalable	Réellement fait	Fait Par
Fenêtre pour visualiser une partie	Itération 1	Itération 1	Grégoire/Tibère
Charger une carte à partir d'un fichier	Itération 1	Itération 1	Damien
Agent aléatoire	Itération 1	Itération 2	Grégoire
Lancer une partie	Itération 1	Itération 1	Adrien/Evan
Modifier la vitesse de simulation	Itération 1	Itération 1	Adrien/Evan/Tibère
Avancer la simulation d'un pas	Itération 1	Pas fait et pas à faire	/
Agent à arbre de décision	Itération 2	Itération 2/3	Adrien/Evan
Sélection du modèle d'agent	Itération 2	Itération 2	Tibère/Grégoire
Sélection de la carte	Itération 2	Itération 2	Tibère/Grégoire
Sélection des paramètres de simulation	Itération 2	Itération 2	Tibère
Sauvegarder/charger une partie	Itération 2	Itération 2	Tibère/Grégoire
Vue de débug (hitbox)	/	Itération	Damien
Lancer un apprentissage génétique	Itération 3	Itération 4	Damien
Paramétrier le réseau de neurone à apprendre	Itération 3	Itération 4	Tibère

Stocker un agent entraîné	Itération 3	Pas fait	/
Affichage spécifique aux entraînements	Itération 3	Pas fait	/
Vue débug (affichage des perceptions)	/	Itération 3	Damien
Amélioration arbre de comportement	/	Itération 3	Evan/Adrien
Refonte du code de l'affichage (Passage de JavaFX à JavaFX - FXML )	/	Itération 3	Grégoire/Tibère
Zoom des cartes, affichage dynamique à la sélection de carte	/	Itération 3	Damien
Lancer un apprentissage par renforcement	Itération 4	Pas fait	/
Choix du type d'apprentissage et des paramètres	Itération 4	Itération 4	Tibère
Éditeur de carte	Itération 4	Itération 3/4	Tibère

# Présentation d'un élément dont nous sommes fier

## Damien – L'affichage de perceptions et l'entraînement

La fonctionnalité que j'ai préféré coder est l'affichage des perceptions, en partie à cause de la praticité de cette fonctionnalité qui nous servira tout au long du projet et qui est très utile pour voir et comprendre ce que les agents perçoivent.

J'aime aussi l'intégration d'ECJ avec toutes les classes que nous avons dû coder durant cette itération, je trouve ça vraiment très satisfaisant de pouvoir voir un réseau totalement incompétent devenir petit à petit de plus en plus performant.

## Adrien – Raycast et moteur

J'ai beaucoup apprécié travailler sur le moteur de jeu, m'assurer que toutes les collisions fonctionnent bien, que la boucle de jeu fonctionne avec tous les éléments existant, prévoir les cas spéciaux. Cela à inclut le développement de rayons pour la vision des agents. Les rayons doivent prendre en compte à la fois les murs et les entités tout en restant suffisamment optimisés pour ne pas trop ralentir le moteur. La difficulté à été d'implémenter deux algorithmes différents, en effet les cases du terrain de jeu sont sur une grille discrète tandis que la position des entités est continue, les optimisations utilisées pour tirer des rayons sur le terrain n'étaient donc pas utilisables pour les entités.

## Grégoire – MVC

J'ai passé beaucoup de temps sur l'étude préalable de la conception de l'engine. Finalement, après quatre itérations, cette conception n'a pas vraiment changé dans sa globalité. Cependant, un aspect qui n'avait pas été réellement anticipé concernait la partie MVC. C'est pourquoi, après deux itérations, nous avons décidé de revoir l'architecture de cette partie, car elle présentait de nombreux problèmes. Le principal problème était l'utilisation d'un unique modèle pour l'ensemble de l'application, alors qu'elle est finalement divisée en trois grandes parties distinctes. De plus, l'écriture du code statique des vues en Java est rapidement devenue problématique. Pour résoudre ce problème, nous avons choisi d'adopter un système basé sur une imbrication des modèles, comme décrit dans la partie [MVC](#) de ce document. Nous en avons également profité pour ajouter plusieurs améliorations :

- L'utilisation de **FXML**, qui permet de rendre la partie statique des pages plus lisible et plus facile à manipuler.
- L'imbrication de vues dans d'autres vues : il est tout à fait possible d'avoir une vue générale avec, par exemple, un **BorderPane** contenant un bandeau en haut, et de charger dynamiquement une autre vue dans la zone centrale du **BorderPane**. Cela

permet d'éviter la multiplication du code du bandeau, qui aurait autrement dû être copié-collé dans chaque vue.

## Tibère – Editeur de carte

La fonctionnalité que j'ai aimé développer est l'éditeur de carte. Elle constitue une partie à part de l'application (avec la simulation et l'apprentissage). J'ai pu développer un chargement et construction de fichier pour créer des nouvelles cartes et modifier des existantes. J'ai aimé développer une fonctionnalité qui est très visuelle et agréable à utiliser. Satisfaisante également car le fait de créer ses propres cartes peut impliquer une certaine créativité.

## Evan – Développement des agents d'arbre de Décision :

La partie que j'ai préféré est celle de la création des agents avec arbre de décision. C'est la fonctionnalité où j'ai pu expérimenter et avancer de la manière la souhaiter dans la façon que les agents se comportent pour obtenir des agents assez compétents tout en observant durant des tests des comportements surprenants et intéressants.

# Planning pour les trois itérations restantes.

L'objectif pour la fin du projet va être de peaufiner notre programme pour le rendre complètement opérationnel et fluide d'utilisation, tout en améliorant les performances de nos agents.

## Itération 5 :

Modifier la distance dans la fonction d'évaluation : distance réelle plutôt que euclidienne  
Sauvegarde automatique de réseau de neurones après apprentissage  
Tests de différents paramètres pour ECJ (recherche d'un apprentissage efficace)  
Utiliser une bibliothèque externe (DL4J) au lieu de MLP pour les réseaux de neurone  
Refactorisation du code  
Co-évolution  
Modifier la fonction d'évaluation dans l'interface  
Apprentissage sur plusieurs cartes

## Itération 6 :

Visualisation d'un réseau de neurone en temps réel  
Ajout d'une perception "mémoire" et affichage associée  
Ajout d'une perception "communication" et affichage associée  
Possibilité pour un humain de jouer  
Entraîner un réseau existant

## Itération 7 :

Objectifs qui n'auraient pas été remplis lors des itérations précédentes  
Résolution des nombreux bugs  
Amélioration esthétique de l'application  
Objets bonus (amélioration de la vitesse de déplacement)

# Conclusion

Pendant ces quatre premières itérations du projet, nous avons découvert de nombreux aspects essentiels.

Tout d'abord, nous avons appris à concevoir un projet de grande envergure et à gérer la complexité qui en découle : structuration du projet, découpage en packages, utilisation de patrons de conception, etc. Nous avons également dû nous adapter à la communication et à la coordination entre les cinq membres du groupe.

Par ailleurs, nous avons appris à utiliser **FXML** afin de mieux structurer notre application et de rendre la création des interfaces beaucoup plus fluide et agréable.

Lors de l'itération 4, nous avons réussi à intégrer **ECJ** à notre projet. Cette intégration nous permettra d'améliorer les capacités d'apprentissage de notre application, notamment pour entraîner des réseaux de neurones avec des configurations très variées, afin de les faire jouer à notre simulateur.

Toutes ces expériences nous ont permis d'apprendre, de progresser et d'évoluer.