



UNIVERSITÉ
DE LORRAINE

IUT nancy Charlemagne
Informatique

Tower defense et Machine Learning

Rapport de fin de projet

Membres : BORTOLOTTI Florian-BOURDON Marin-DUCHÈNE Éloi-ROTH Tristan

Tuteur : Vincent THOMAS

Année : 2024/2025

Table des matières :

Introduction	3
Objet du document	3
Présentation du projet	3
Présentation de l'équipe et rôle de chacun	5
Tristan	5
Marin	5
Florian	6
Éloi	6
Planning de déroulement du projet	7
Itération 1 :	7
Itération 2 :	8
Itération 3 :	9
Itération 4 :	10
Itération 5 :	10
Itération 6 :	11
Itération 7 :	11
Analyse	11
Modèles UML utilisés	11
Évolution de notre projet	15
Réalisation	18
Architecture logicielle	18
Evolution	19
Test de validations	21
Difficultés rencontrées	27
Planning	27
Itération 1	27
Itération 2	28
Itération 3	28
Itération 4	29
Itération 5	29
Itération 6	30
Itération 7	30
Conclusion	30
Annexe	32
Installer depuis GitHub	32
Lancer le projet	33

Introduction

Objet du document

Dans le cadre de notre projet tutoré de dernière année en BUT Informatique, nous avons entrepris le développement d'un jeu de type *Tower Defense*, avec une approche exploratoire axée sur l'apprentissage automatique. Contrairement aux jeux traditionnels où le joueur doit défendre son territoire contre des vagues d'ennemis prévisibles, notre projet met l'accent sur l'évolution et l'adaptation des ennemis. L'objectif principal est de concevoir un système dans lequel les ennemis évoluent en fonction de leurs performances et ajustent leurs stratégies afin d'exploiter les failles des défenses mises en place par le joueur.

Pour atteindre cet objectif, nous intégrons des techniques d'apprentissage automatique permettant aux ennemis de modifier leurs caractéristiques (vitesse, vie, attaque, etc.) en fonction de leur survie et de leurs performances durant chaque manche. Nous utilisons plusieurs algorithmes, notamment les algorithmes évolutionnaires, l'algorithme de choix de chemin A* couplé au Steering Behavior, afin de modéliser un comportement dynamique et intelligent des adversaires.

Ce rapport détaille notre démarche, depuis l'analyse du projet et la modélisation jusqu'à la réalisation technique et l'évaluation des performances de notre système. Nous présentons également l'organisation du travail en équipe, les choix technologiques, ainsi que les difficultés rencontrées et les solutions mises en place. Enfin, nous parlons des différents objectifs à atteindre pour les prochaines itérations.

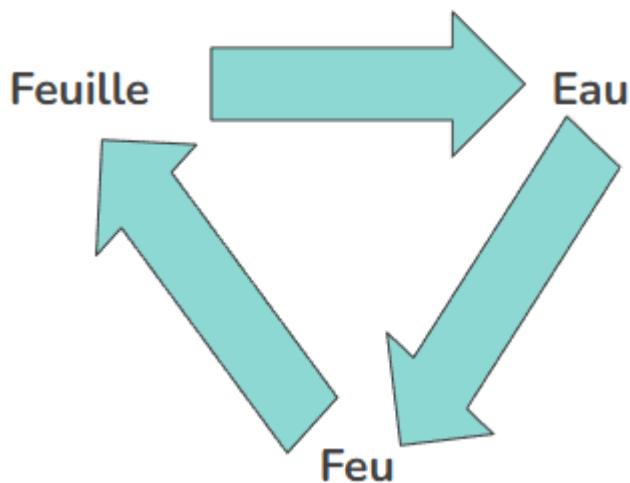
Présentation du projet

Pour bien comprendre le projet dans ses détails, notamment l'évolution, il est important de rappeler le **fonctionnement de nos entités**, qui possèdent plusieurs caractéristiques qui les définissent :

Le type :

Les types sont les suivants : feu, eau, feuille.

L'ordre d'efficacité peut être représenté par ce graphique :



Si le type est un type plus fort, l'attaque voit ses dégâts augmentés. Si le type est un type plus faible, l'attaque voit ses dégâts diminués. Si les deux types sont les mêmes, l'attaque ne voit pas ses dégâts modifiés.

+ Les comportements :

Les comportements sont les suivants : Normal, Fuyard, Healer, Kamikaze

Ils définissent pour un ennemi la manière de se déplacer :

Normal : passe au chemin le plus court, sans considérer les tours.

Fuyard : évite les tours, sauf obligation.

Kamikaze : passe au chemin le plus court, en attaquant la première tour rencontrée.

Soigneur : prend le chemin pris par le plus d'ennemis et fait un soin de zone.

Genre	Comportement	Sprite en mode "normal"	Sprite en mode "simple"
Géant	Normal		
Ninja	Fuyard		
Druide	Healer		
Berserker	Kamikaze		

L'image ci-dessus résume le comportement associé à chaque ennemi, ainsi que ses sprites dans les différents modes d'affichage du jeu.

Il en va de même pour nos défenses, la seule différence étant que les défenses n'ont pas de comportement spécifique, et se contentent d'attaquer un ennemi

jusqu'à ce que celui-ci soit mort ou hors de portée. Ci-dessous une image présentant les sprites associés à chaque défense :

Genre	Sprite en mode "normal"	Sprite en mode "simple"
Archer		
Canon		

Présentation de l'équipe et rôle de chacun

Tristan

Au cours de l'itération 1, Tristan a réalisé le prototype du moteur de jeu ainsi qu'une première interface graphique. Il a continué sur cette lancée lors de l'itération 2, en participant à la fusion entre l'algorithme de déplacement des ennemis et l'interface graphique. Il a également ajouté un panneau de log et des sprites à l'interface graphique. Lors de l'itération 3, il a implémenté la fonctionnalité permettant de lancer le jeu sans interface graphique. Pour ce qui est de l'itération 4, Tristan a réalisé l'évolution d'un unique ennemi (évolution se focalisant sur l'évolution des statistiques de l'ennemi). Lors de l'itération 5, Tristan a amélioré la fonction de score et l'interface, notamment grâce à la génération de graphiques pour suivre l'évolution de la population à chaque manche et la vie de chaque ennemi en temps réel. Durant l'itération 6, il a réalisé une première version de l'évolution basée sur Steering behaviors, qu'il a ensuite complétée durant l'itération 7 pour obtenir une évolution sur le mouvement fonctionnelle.

Marin

Marin a réalisé le prototype de l'algorithme évolutionnaire lors de l'itération 1. Il a ensuite travaillé à améliorer cet algorithme au cours des itérations 2 et 3, notamment au niveau du calcul de la fonction de score. Il a également implémenté l'attaque des ennemis sur les défenses lors de l'itération 2. Il a réalisé à l'itération 4 l'évolution d'un groupe d'ennemis (évolution visant à faire évoluer la composition du groupe d'ennemi, en changeant les proportions de chaque type d'ennemi dans le groupe). Durant l'itération 5, il a amélioré l'interface graphique en ajoutant un "mode simple",

qui est un affichage du jeu sans les sprites, bien plus épuré et simple à comprendre que l'affichage classique. Pour l'itération 6, Marin a ajouté une fenêtre de chargement lors de l'évolution, ainsi qu'un écran de fin de partie. Lors de l'itération 7, il a implémenté la possibilité de recommencer une partie directement depuis l'écran de fin de partie.

Florian

Florian a réalisé au cours de l'itération 1 le prototype de Steering behaviors, qu'il a ensuite fusionné lors de cette même itération à l'algorithme de déplacement A*. Il a participé durant l'itération 2 à la fusion entre l'algorithme de déplacement et l'interface graphique. Au cours de l'itération 3, il a ajouté un comportement d'évitement des obstacles à Steering behaviors, et à refactor le code de la classe Entities et des classes filles de celle-ci. Lors de l'itération 4, il a participé à l'implémentation d'un déplacement basé uniquement sur Steering behaviors. Pour l'itération 5, il a ajouté la possibilité de modifier la vitesse du jeu, ainsi qu'un nouveau comportement, l'arrival behavior. Durant l'itération 6, Florian a effectué une refonte graphique du menu de démarrage de la partie, et ajouté des configurations pré-faites pour démarrer la partie. Lors de l'itération 7, il a ajouté les sprites des défenses en mode simple.

Éloi

Lors de l'itération 1, Éloi a réalisé le prototype de l'algorithme A*, qu'il a ensuite fusionné lors de cette même itération à Steering behaviors. Il a participé durant l'itération 2 à la fusion entre l'algorithme de déplacement et l'interface graphique. Au cours de l'itération 3, il a ajouté des sprites aux différents ennemis sur l'interface graphique, et a implémenté un recalcul de A* lors de la destruction d'une tour. Lors de l'itération 4, il a participé à l'implémentation d'un déplacement basé uniquement sur Steering behaviors. Pour l'itération 5, il a ajouté la possibilité de sélectionner un labyrinthe non répertorié, et ajouté un nouveau comportement, l'arrival behavior. Durant l'itération 6, Éloi a créé un bouton "aide", qui donne des informations sur le jeu, et a refondé graphiquement le menu de démarrage. Il a également créé une fenêtre séparée pour l'affichage des différents graphiques. Lors de l'itération 7, il a modifié les sprites des défenses, ajouté les sprites des défenses en mode simple et ajouté un affichage du nombre d'ennemi restant pour atteindre l'objectif mis à jour en temps réel.

Il est bon de noter ici qu'à partir de l'itération 2, chaque membre du projet a effectué beaucoup de bugfix. En effet, la mauvaise structure de notre projet engendre de nombreux bugs, ce qui fait que le temps passé à les résoudre est assez important.

De plus, les itérations 6 et 7 apportent moins de nouvelles fonctionnalités que les itérations précédentes, en raison de leur durée plus faible que les autres.

Planning de déroulement du projet

Nous allons présenter ci-dessous le planning initialement prévu. Celui-ci a changé au cours du projet, car nous avons dû nous adapter aux différences entre nos avancées prévues et réelles. Le planning réel de ce qui a été effectué au cours du projet sera présenté plus loin dans ce document.

Itération 1 :

- Implémentation du moteur de jeu. On peut lancer une partie avec différents ennemis et différentes défenses via une popUp en début de partie permettant de choisir la population initiale. Toutes ces entités sont clairement représentées sur le labyrinthe.
- Prototypage pour les steering behaviors
- Prototypage de l'algorithme évolutif.
- Prototypage de l'algorithme A*.

Critères de validation :

- Les prototypes sont fonctionnels
- Création des éléments que nous retrouverons dans le jeu (défenses, ennemis, labyrinthe).

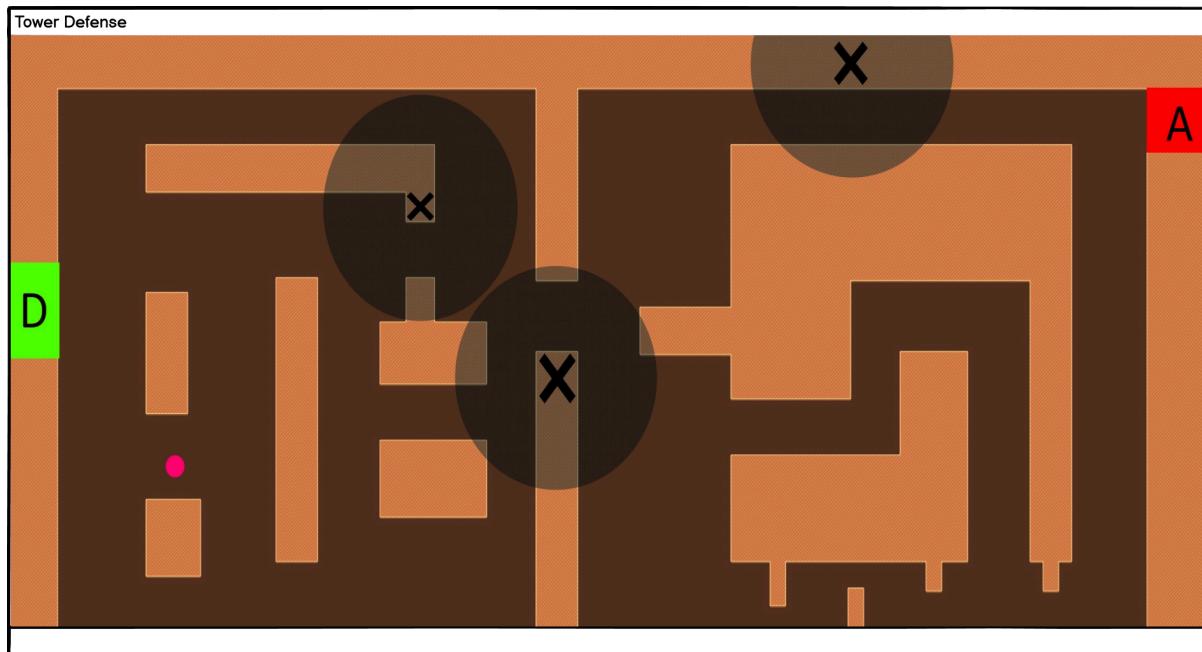
Critères de validation :

- Les ennemis peuvent se déplacer, les tours sont présentes avec leur zone, le labyrinthe se génère.

Répartition des tâches :

- Éloi → A*
 - Florian → Algorithme Steering Behaviors
 - Marin → Algorithme évolutif
 - Tristan → Moteur de jeu et interface graphique
 - Ceux qui termineront leur prototype en 1^{er} pourront se mettre sur la création des éléments (défenses, ennemis, labyrinthe).
-
- **Produit Minimum Viable :** Les quatre prototypes sont fonctionnels et nous pouvons générer une carte à partir d'un fichier .txt avec un ennemi qui peut se déplacer de manière autonome, mais limité.

Maquette :



Itération 2 :

- Fusion des prototypes de l'itération 1.

Critère de validation :

- Les prototypes fonctionnent les uns avec les autres.
- Ajout d'un panneau latéral contenant les informations du déroulement de la partie.

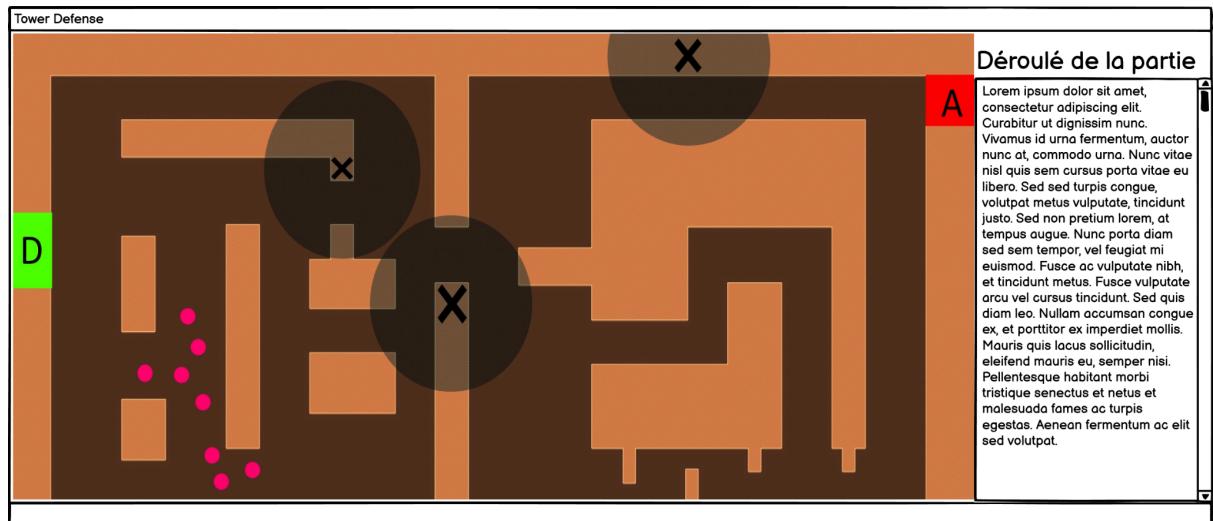
Critère de validation :

- Les informations affichées sont concordantes avec les évolutions choisies

Répartition des tâches :

- Éloi → Fusion
- Florian → Fusion
- Marin → Fusion
- Tristan → Fusion
- Ceux qui auront terminé la fusion en premier feront le panneau latéral.
- **Produit Minimum Viable :** Les ennemis peuvent se déplacer de manière autonome en suivant un chemin défini par l'algorithme A*. Ils évoluent entre les différentes vagues via l'algorithme évolutif. Ils évoluent seulement sur leurs caractéristiques intrinsèques (vitesse, vie).

Maquette :



Itération 3 :

- Créations de types d'ennemis et de défenses
- **Critères de validation :**
 - Les dégâts subis sont bien modifiés selon la relation des types (x1.5 si bon type attaquant, x0.5 si mauvais type attaquant, x1 sinon)
- Création des comportements, qui vont complexifier les mouvements des ennemis.

Critères de validation :

- Un comportement modifie bien le choix du chemin adopté par l'ennemi.
- Ajout des sprites.
- Les tours peuvent tirer lorsque les ennemis sont dans leur zone de tir.

Critères de validation :

- Les dégâts des tirs sont pris en compte avec le type de la tour et de l'ennemi.
- Une tour ne peut attaquer qu'un seul ennemi à la fois
- Une tour attaque le même ennemi tant que celui-ci n'est pas mort, ou est sorti de sa portée d'attaque.
- Lors de son changement de cible, elle prend pour cible en priorité l'ennemi le plus proche de l'arrivée.

Répartition des tâches :

- Éloi → Sprites / Types
- Florian → Comportement
- Marin → Tir des tours
- Tristan → Tir des tours

- **Produit Minimum Viable** : Les ennemis ainsi que les tours peuvent avoir des types. Les ennemis évoluent également sur les types et leur comportement.

Maquette:



Itération 4 :

- Nous gardons l'itération 4 pour résoudre les éventuels problèmes des itérations précédentes (exemple : hit box des ennemis, passage par un chemin qu'une défense ne couvre pas totalement). Si plus aucun problème, améliorer l'évolution des ennemis en changeant la diversité des vagues d'ennemis de manche en manche (exemple : on retire des géants pour les remplacer par des ninjas).
- Préparation de la soutenance
- **Produit Minimum Viable** : Déplacement et évolution fonctionnels

À partir de l'itération 5, nous allons commencer la “course à l'armement”, qui consiste à faire évoluer les défenses en même temps que les ennemis.

Itération 5 :

- Adaptation de l'algorithme d'évolution pour les tours.
- Ajouts des critères de victoires et de défaites.
 - Victoire des ennemis :
 - Un certain nombre d'ennemis passent l'arrivée dans un certain nombre de vagues.
- **Produit Minimum Viable** : Défenses adaptatives

Itération 6 :

- Résolution des erreurs et ajout des fonctionnalités pas encore pensées.
- **Produit Minimum Viable** : Produit final

Itération 7 :

- Préparation de la soutenance finale et du document final
- **Produit Minimum Viable** : Version finale avec toutes les fonctionnalités

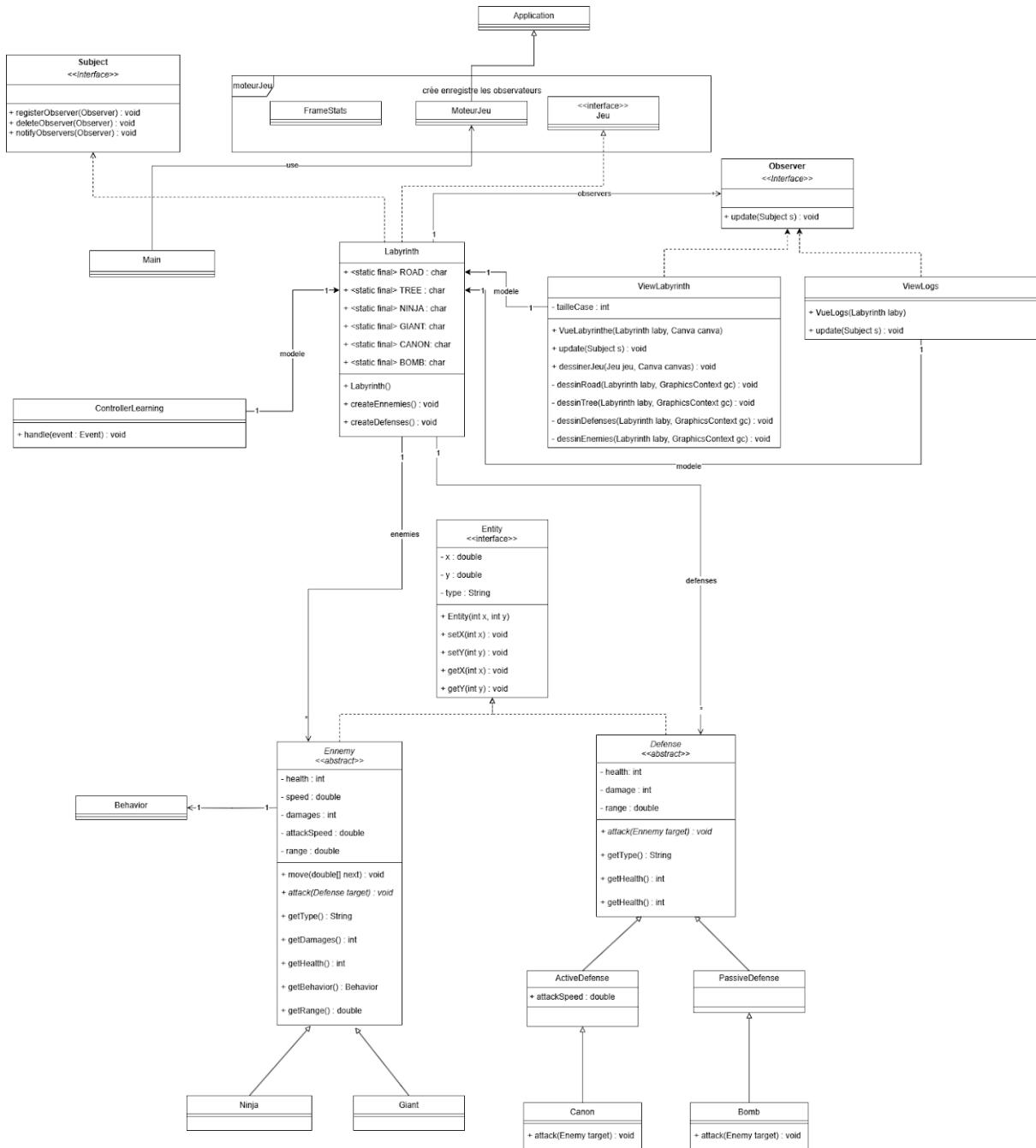
Analyse

Modèles UML utilisés

Au cours de notre étude préalable, en réfléchissant sur l'architecture de notre projet et après des discussions avec notre tuteur de projet, nous avons décidé de commencer par créer des **prototypes** des gros blocs de notre Tower Defense, ces prototypes étant les suivants :

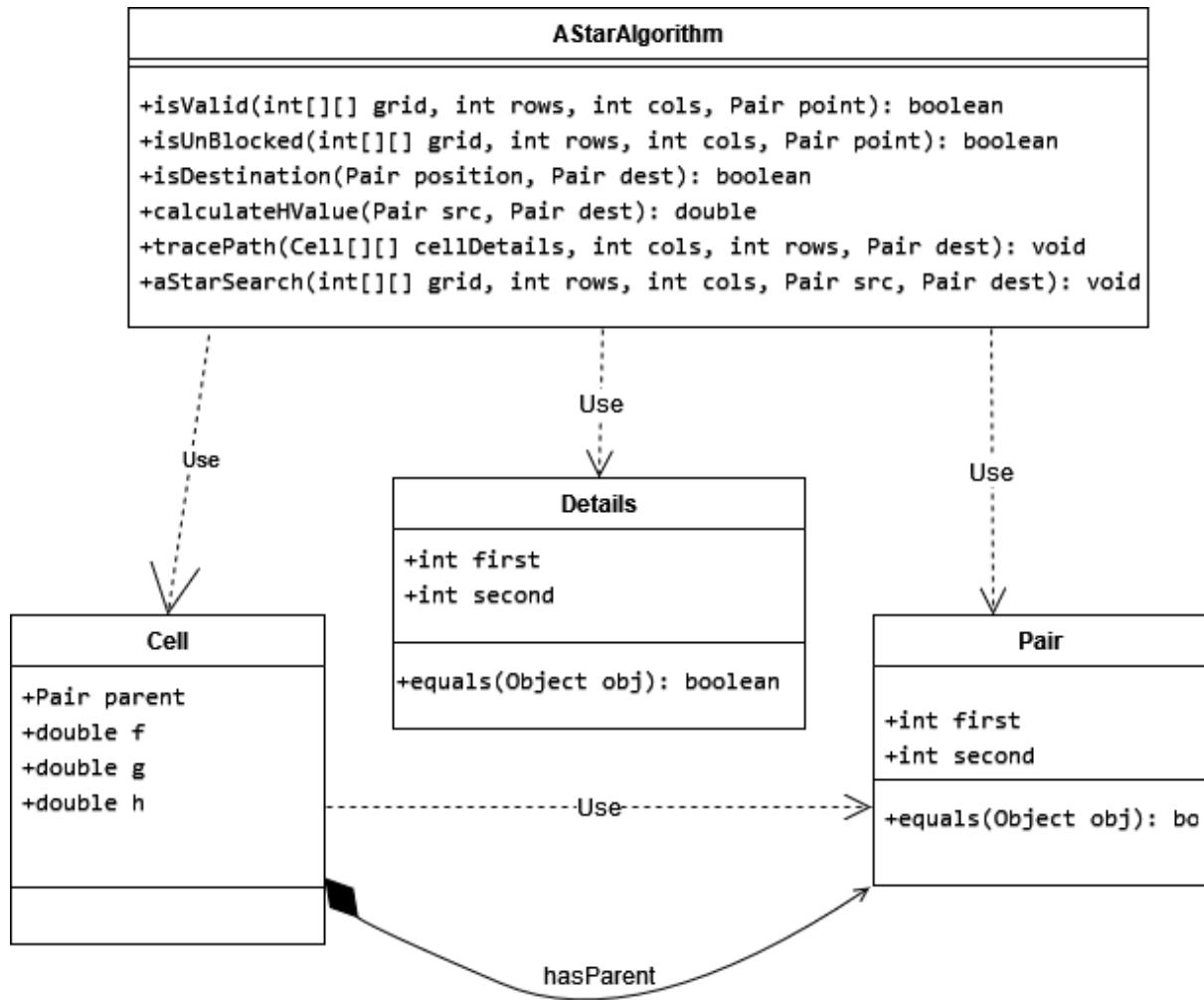
Tout d'abord, nous avons développé le prototype du **moteur de jeu et de l'interface graphique**. Le déroulement d'une partie est rendu possible grâce au moteur de jeu, qui gère le temps et les interactions entre les différents agents du jeu, il est basé sur Zeldiablo (projet du Semestre 2). Nous l'avons ajusté à notre architecture initiale réalisée en respectant le patron de conception MVC, celle-ci permettant de définir les ennemis, les défenses ainsi que les composants graphiques. Le produit minimal viable qui en résulte permet de faire tourner le jeu en créant un labyrinthe composé d'ennemis et de défenses, configuré via une interface utilisateur au lancement du programme, et représenté graphiquement.

Diagramme de classe correspondant :



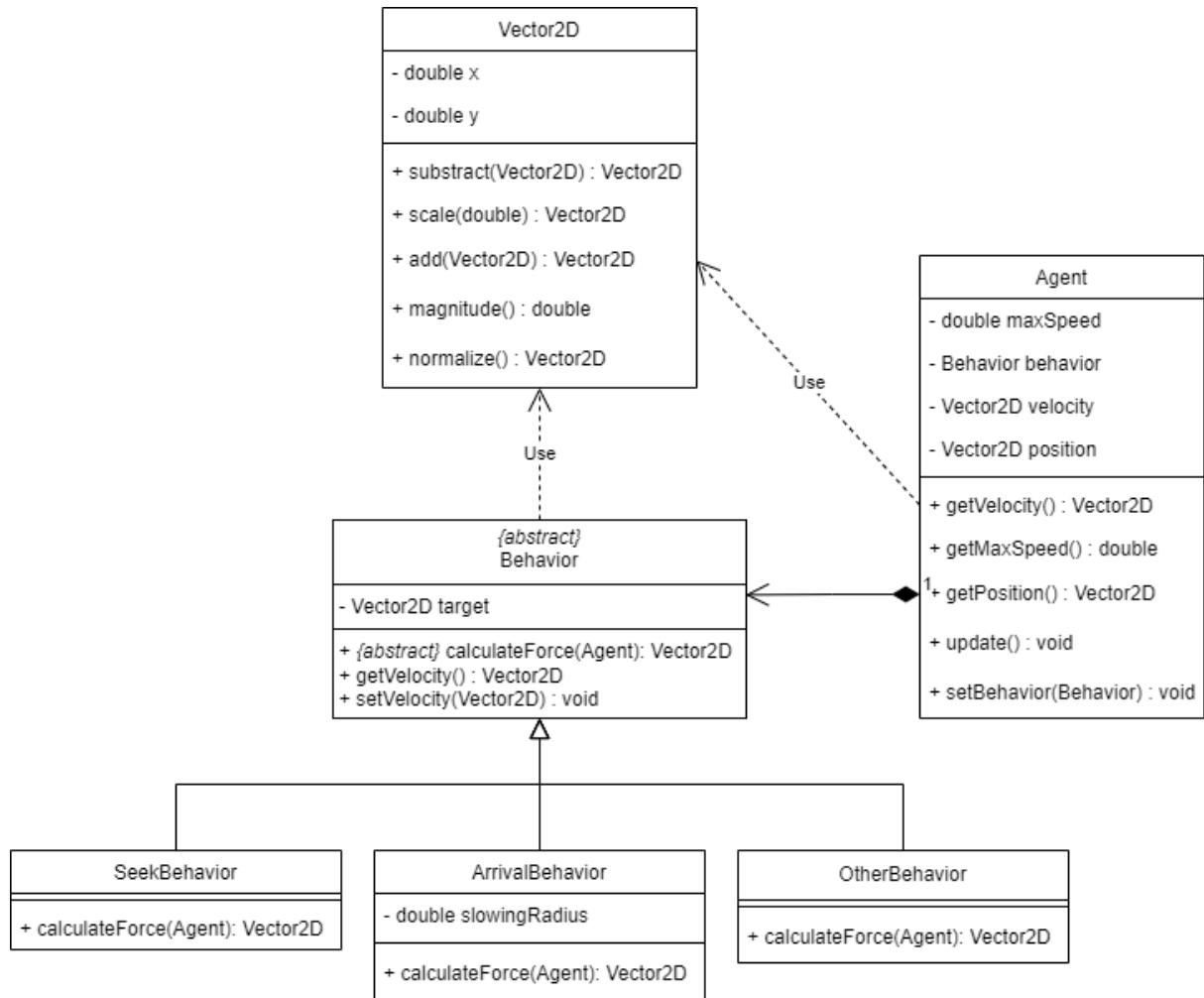
Ensuite, nous avons le prototype de **l'algorithme de choix de chemin**, pour ce prototype, nous avons décidé d'utiliser l'algorithme A*. Il utilise une évaluation heuristique sur chaque nœud d'un graphe pour estimer le meilleur chemin du nœud initial au nœud final, il visite ensuite les nœuds de ce chemin.

Diagramme de classe correspondant :



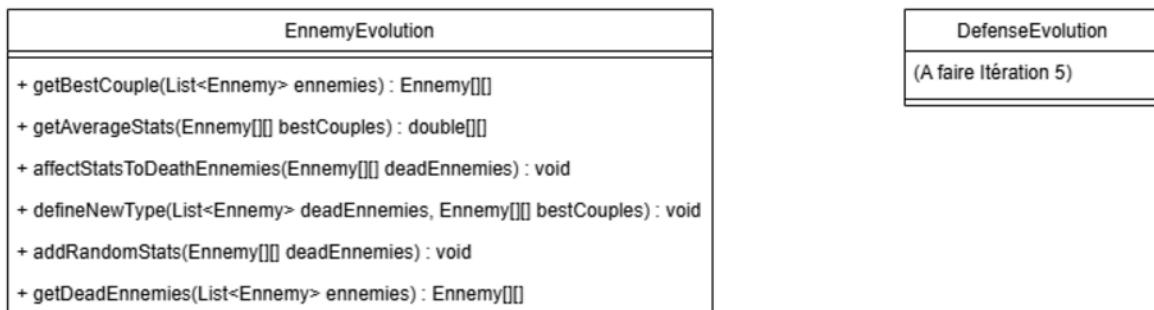
Nous avons également réalisé le prototype d'un **algorithme de déplacement**, représenté par un Steering Behavior, son comportement permet de simuler des mouvements réalistes en combinant plusieurs forces influençant la direction et la vitesse d'une entité. Le comportement de base inclut le suivi de chemin calculé par A*.

Diagramme de classe correspondant :



Pour finir sur les prototypes, nous avons celui de l'algorithme d'évolution :

Diagramme de classe correspondant :



Évolution de notre projet

Le fonctionnement de notre projet a cependant bien changé depuis notre première phase d'analyse, tout d'abord, nous avons **fusionné** les deux algorithmes gérant le déplacement, soit **A*** et **Steering Behavior**.

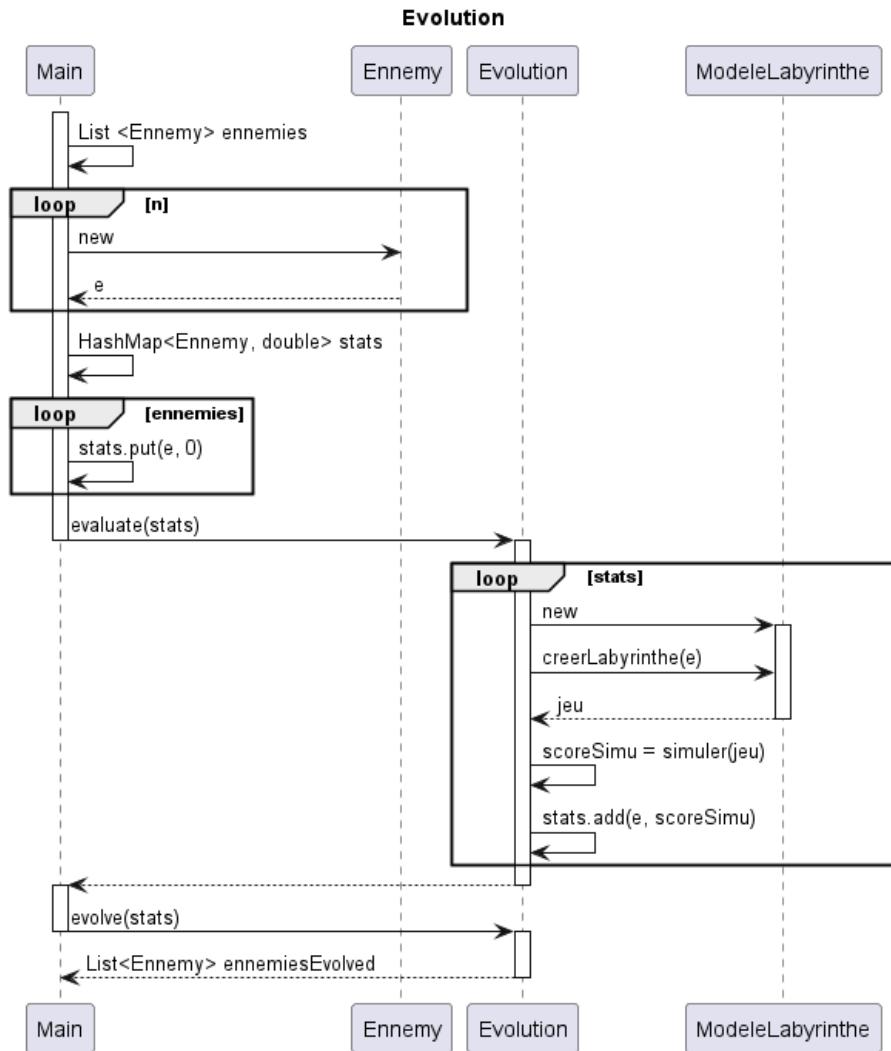
Diagramme de classe correspondant :



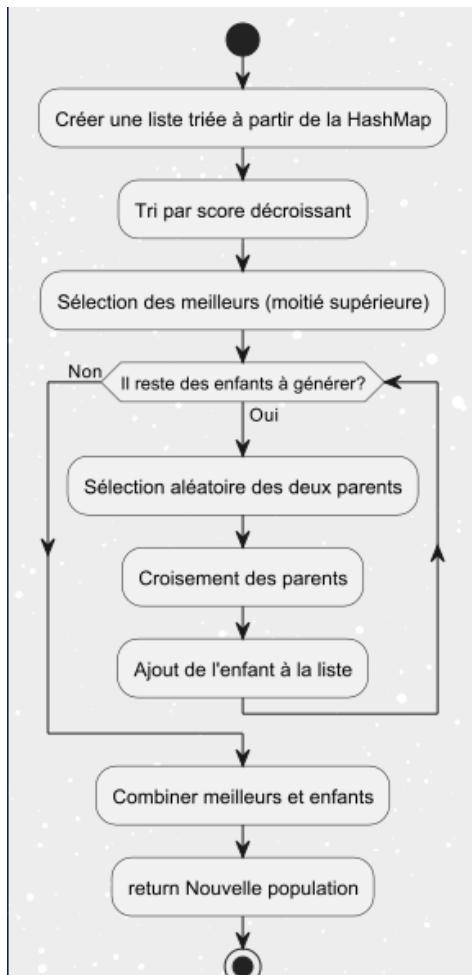
Cette fusion nous a permis de nous rendre compte que le déplacement des ennemis se basait surtout sur A* et qu'il ne pouvait donc pas vraiment avoir d'apprentissage sur le déplacement d'un ennemi. Nous avons cependant continué sur cette approche qui est tout de même intéressante en calculant un chemin différent en fonction de l'ennemi. En effet, comme expliqué ci-dessus, le comportement de l'ennemi définit les contraintes relatives au choix du chemin de celui-ci par A*.

Nous utilisons tout de même **Steering Behavior** et avons décidé de l'utiliser d'une autre approche, il est maintenant possible de le rendre indépendant de A* avec un déplacement de l'ennemi en fonction des forces exercées. L'objectif pour la suite était donc de pouvoir apprendre les trajectoires des ennemis en faisant évoluer leur liste de checkpoints.

Notre approche par rapport à **l'évolution** des ennemis a, elle aussi changée. Nous avons repris l'évolution du début après avoir rencontré des problèmes pour l'intégrer à nos autres prototypes, notre évolution est maintenant représentée par ce diagramme de séquence :



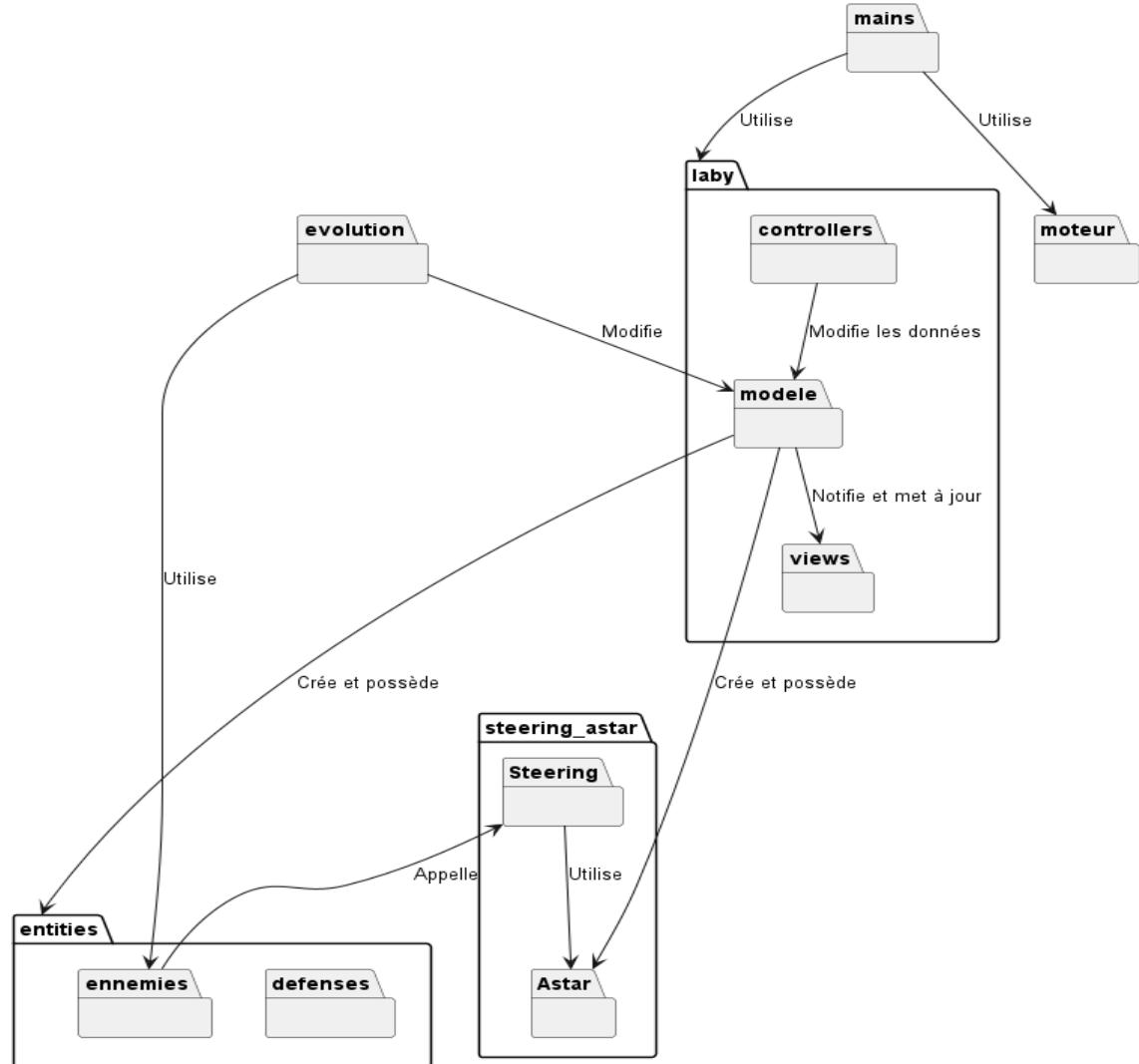
Avec une fonction evolve décrite par ce diagramme d'activité :



Enfin, nous n'avons pas réalisé l'évolution des tours, par **manque de temps**. En effet, l'évolution des tours aurait été un sujet intéressant, mais nous avons jugé que développer l'évolution sur le **mouvement des ennemis**, c'est-à-dire l'évolution sur `Steering behavior` serait plus intéressant, puisqu'il amenait un nouveau type d'évolution. Nous avons donc remplacé l'évolution des tours par l'évolution de `Steering behavior`.

Réalisation

Architecture logicielle



On peut voir sur le schéma ci-dessus la structure globale de notre application. On y retrouve notamment les prototypes réalisés durant l'itération 1 : le prototype du moteur de jeu correspond au bloc "laby", le prototype de l'évolution à l'élément "evolution", et enfin les prototypes de A* et Steering behaviors sont dans le bloc "steering_astar", bloc qui regroupe les éléments chargés du déplacement des entités. En parallèle de cela, on peut trouver le bloc entities qui regroupe les ennemis et les défenses.

Pour ce qui est des interactions entre ces différents blocs, le modèle créé des entités. Les ennemis vont utiliser Steering behaviors pour se déplacer le long de chemins définis par A*.

Evolution

Au cours de notre projet, nous avons intégré de l'évolution comme le voulait notre sujet. Après notre phase d'analyse nous avons décidé que l'évolution des ennemis se fera tout d'abord sur leurs **statistiques**, puis sur leur **effectif** et quand tout ça sera fonctionnel ce sera sur leur **déplacement** également que les ennemis évoluent.

Nous avons donc créé un premier prototype qui avait simplement pour objectif de maximiser les statistiques des ennemis. Cette base nous a servie pour toutes les évolutions suivantes, on retrouve une classe Evolution qui possède plusieurs méthodes, une première méthode **évaluation** qui prend une HashMap en paramètre ayant pour clef un groupe d'ennemis et pour valeur un score (le groupe est composé d'un seul ennemi quand nous ne voulons pas faire évoluer un groupe). Cette HashMap sera initialisé dans le ContrôleurLearn ou dans un main pour tester, le score de chaque groupe d'ennemi est initialisé à 0 au moment de l'appel de la méthode évaluation, cette méthode aura justement pour but de mettre à jour cette HashMap en calculant le score de chaque groupe d'ennemis. Pour ce faire, elle fait appel à la méthode **simulation** qui simule une partie avec le groupe en paramètre et ensuite la méthode **getScore** qui prend en paramètre un groupe d'ennemis simuler pour renvoyer son score. La fonction de score est le cœur de l'évolution, on définit nos attentes via les paramètres et valeurs que nous allons utiliser dans cette fonction pour renvoyer un score. On retrouve ensuite la fonction **evolve** qui prend en paramètre une HashMap d'ennemi qui doit être évalué. Cette fonction commence par découper la HashMap en deux pour ne garder que les meilleurs groupes, elle crée ensuite des groupes enfants grâce à la méthode **croiser** à qui prend en paramètre deux parents tirés au hasard parmi les meilleurs groupes. La méthode de croisements est elle aussi différente en fonction du type d'évolution que nous réalisons. La méthode de croisement nous permet donc de créer un nombre d'enfants égal au nombres de groupes que nous avons abandonné pour ne garder que les meilleurs, une fois l'obtention de notre nouvelle population, nous la passons dans la méthode **mutate** qui permet, elle aussi en fonction du type d'évolution, d'ajouter du bruit sur ce qu'on fait évoluer. On prend ensuite le groupe ayant obtenu le meilleur score à la manche précédente pour jouer la manche suivante via l'interface graphique, on répète cette opération pour chaque manche et obtient donc des groupes à chaque fois meilleurs.

Evolution sur les statistiques :

- Fonction de score : utilise la vie de l'ennemi, la distance à l'arrivée de l'ennemi ainsi que le temps de survie de l'ennemi pour calculer le score. On ajoute à toute ces valeurs un bonus très élevé en fonction de si l'ennemi est mort ou non.

- Croisement : on prend une statistique au hasard d'un parent et la donne à l'enfant, et ça pour toutes les statistiques.

- Mutation : on fait muter de 5% chaque statistique de l'ennemi soit vers le bas soit vers le haut, ce choix est fait de manière aléatoire.

Evolution de l'effectif :

L'évolution du groupe d'ennemis utilisera le groupe existant ainsi que les autres groupes que le programme évaluera à un moment donné, lorsque l'utilisateur appuie sur le bouton "learn". À partir de ces groupes d'ennemis, l'algorithme d'évolution va sélectionner les meilleurs et créera des enfants à partir de ceux-ci. Un groupe enfant est le résultat du croisement de deux parents tirés au hasard parmi les meilleurs. On choisit ensuite des agents au hasard parmi ces groupes. Après avoir obtenu la nouvelle population de groupes d'ennemis, une mutation est appliquée sur les statistiques de chaque ennemi. Enfin, le meilleur groupe est envoyé dans le jeu.

- Fonction de score : appelle la fonction de score sur les statistiques de chaque ennemi du groupe et on fait la somme de ces scores ce qui nous donne le score du groupe.

- Croisement : Les deux parents étant des groupes d'ennemis avec un effectif différents, le groupe enfant prend à chaque fois un ennemi au hasard de chaque groupe pour le mettre dans le groupe enfant, on ne peut pas tirer deux fois le même ennemi, si c'est le cas on en tire un nouveau.

- Mutation : on réalise la même mutation que pour les statistiques.

Nous nous sommes rendu compte que le fait de faire muter les statistiques et de garder les meilleurs ennemis grâce à la fonction de score qui est plus élevé si les ennemis ont des meilleures statistiques est en fait une évolution sur les statistiques des ennemis en plus de l'évolution de l'effectif du groupe d'ennemis.

Evolution du déplacement :

L'évolution du Steering Behavior fonctionne de la même manière, la différence étant que nous n'utilisons pas des groupes d'ennemis, mais un seul ennemi, qui est un géant. Ainsi, après avoir obtenu une population de groupes de géants, nous sélectionnons les meilleurs, ayant la trajectoire la mieux évaluée pour créer des enfants. Le croisement fonctionne de la manière suivante : on prend le premier waypoint des deux parents et on donne à l'enfant le milieu de ces deux waypoints, puis on répète ce processus pour les waypoints suivants. On ajoute ensuite de la mutation sur chaque waypoints de tous les ennemis (déplace un petit peu).

L'évolution du déplacement utilise uniquement Steering Behavior et abandonne A*, on apprend donc maintenant les waypoints que l'ennemi parcourt pour atteindre l'arrivée, on les crée aléatoirement pour chaque ennemi et on essaye d'obtenir la meilleure combinaison de waypoints pour un seul ennemi.

- Fonction de score : utilise la distance parcourue de l'ennemi et le bonus très élevé de si l'ennemi est arrivé ou pas, ça nous permet dans un premier temps de garder les ennemis étant arrivés et ensuite ceux qui sont arrivés le plus vite possible donc avec la meilleure trajectoire.

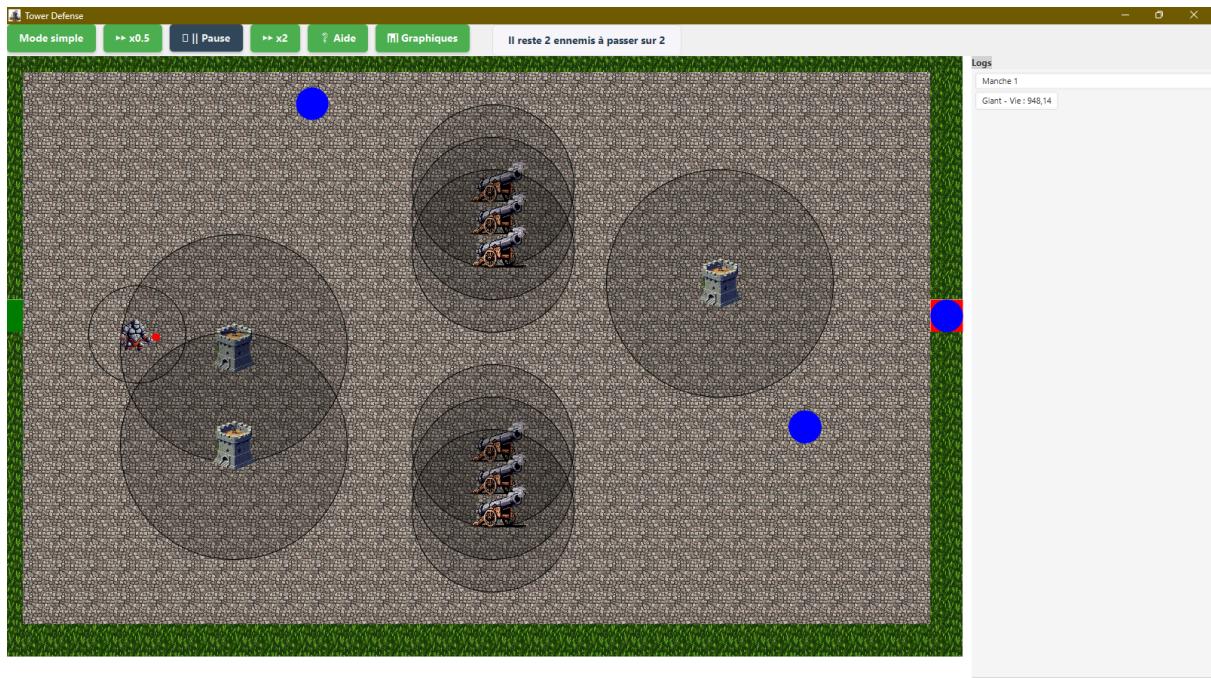
- Croisement : on prend le premier waypoint des deux parents et on donne à l'enfant le milieu de ces deux waypoints, puis on répète ce processus pour les waypoints suivants.

- Mutation : on fait muter les waypoints pour créer de la diversité et ainsi maximiser nos chances d'obtenir une solution optimale.

Test de validations

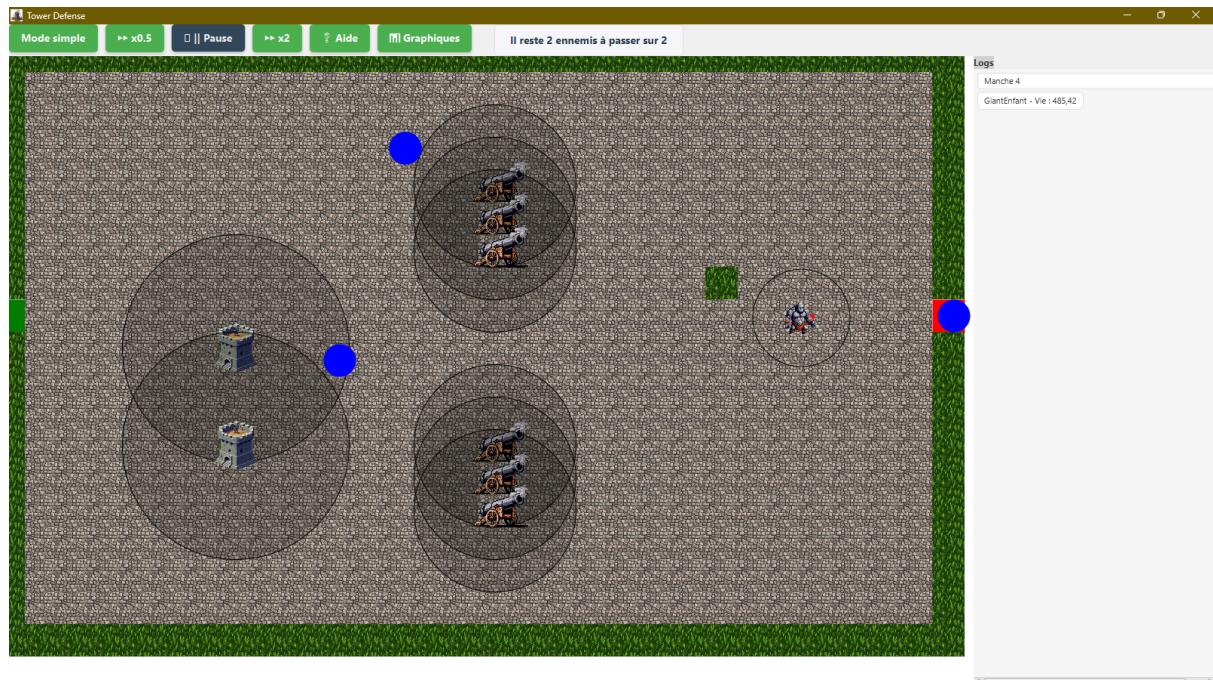
Evolution Steering Behavior

Manche 1 :



On voit qu'à la première manche, les waypoints de l'ennemi sont bien aléatoires, le premier est à droite du labyrinthe et le deuxième en haut à gauche. L'ennemi va donc forcément mourir, car sa liste de waypoints est trop aléatoire.

Manche 4 :



On voit qu'à partir de la manche quatre, on obtient déjà une trajectoire correcte qui permet à l'ennemi d'atteindre l'arrivée Ici l'ennemi a pour premier waypoint un qui est placé en haut pour éviter les premières tours et son deuxième waypoint est au centre pour pouvoir atteindre l'arrivée sans passer sur un canon.

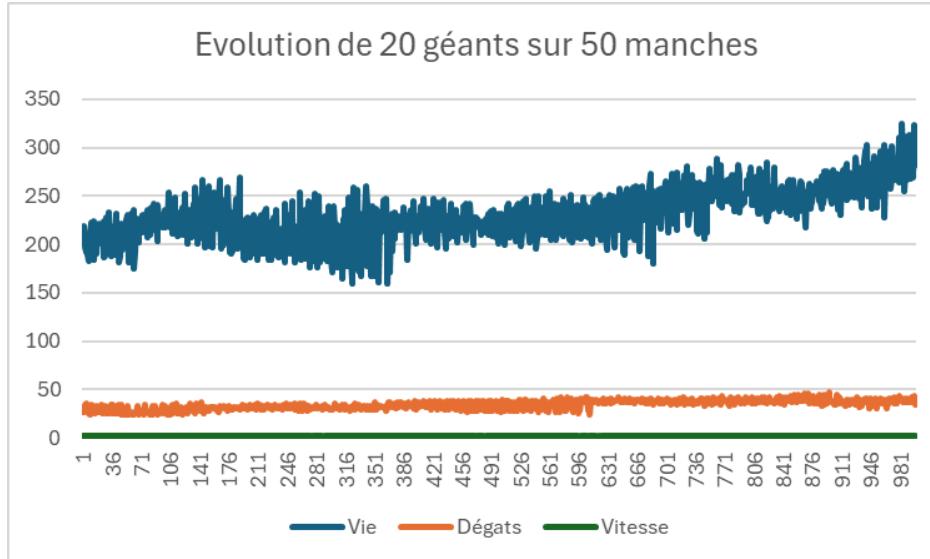
Manche 5 :



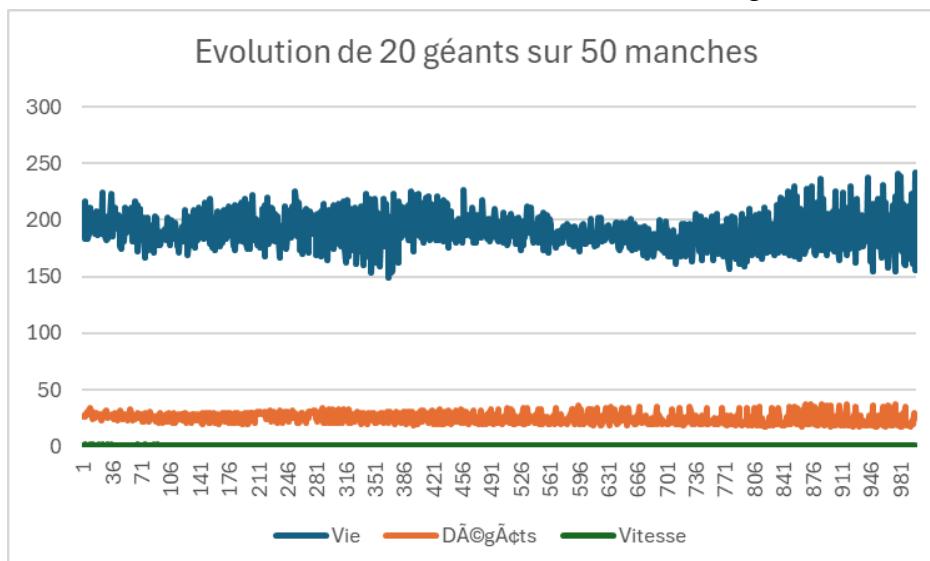
A la manche 5, qui est la manche suivante, on voit que l'on trouve une nouvelle solution qui est logiquement encore plus optimisée que la précédente. On a donc au moins deux solutions viables assez rapidement.

Evolution sur l'effectif

Évolution pour 1 individu :

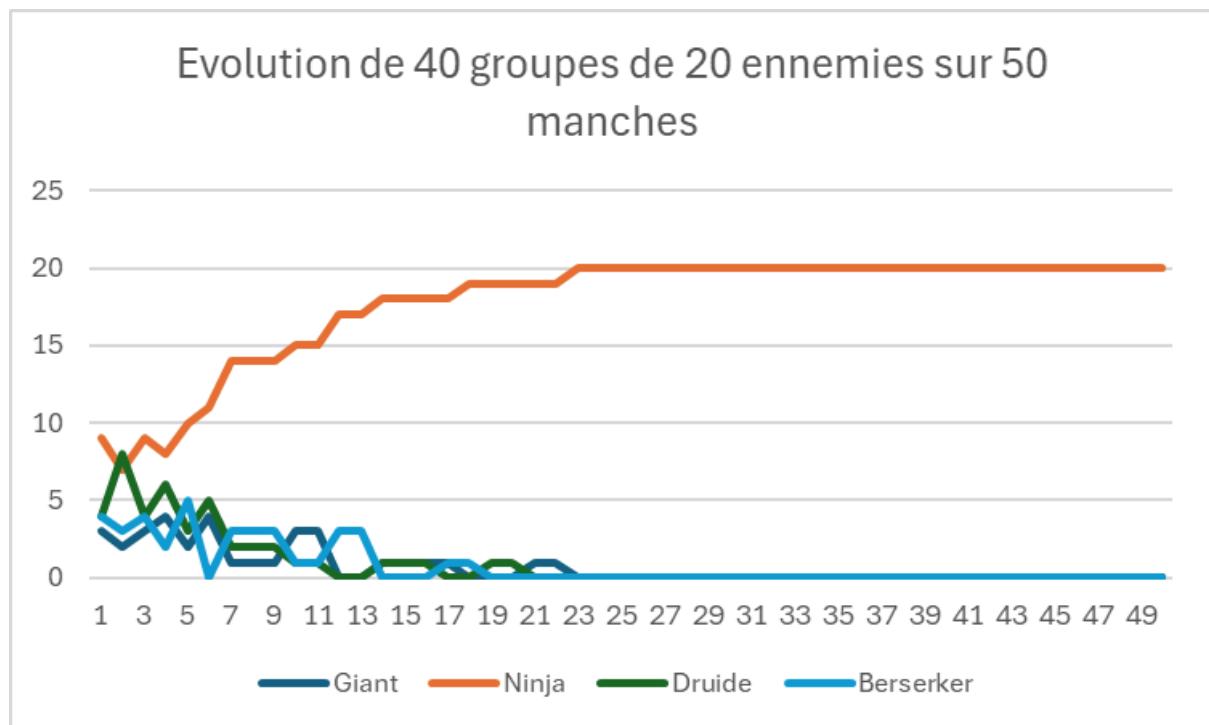


On voit que la courbe de vie diverge ici vers le haut et la vie augmente bien, L'axe des ordonnées représente la valeur réel de l'attribut de l'ennemi, et l'axe des abscisses représente lui le nombre de manche, ici, il y en a bien 50 mais les valeurs incorrect sont dû une mauvaise création de graphique, en effet ici on affiche à chaque manche la liste des ennemis évoluer et pas seulement le meilleur ennemi de la manche, nous allons améliorer notre création de graphique pour 1 ennemi par la suite. Le fait d'afficher la liste est aussi la cause des grosses ondulations.



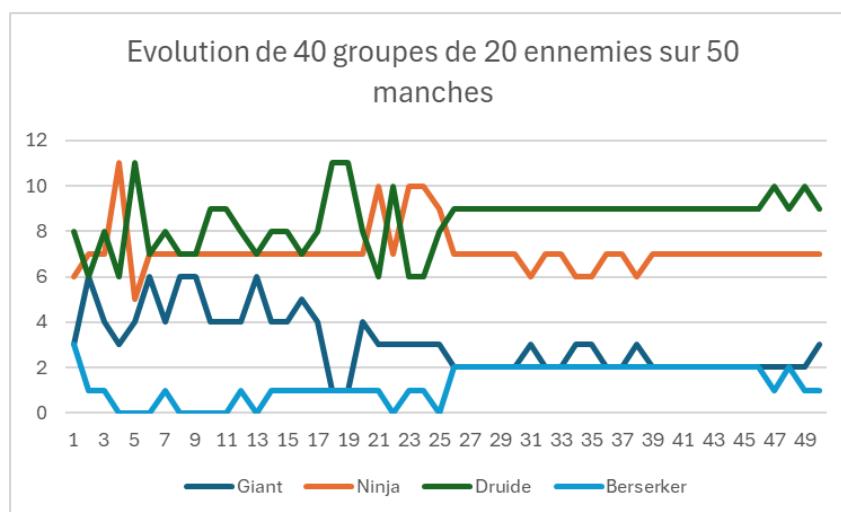
Ici l'exemple d'une autre exécution où les statistiques ont stagné.

Évolution pour 1 groupe d'individus :



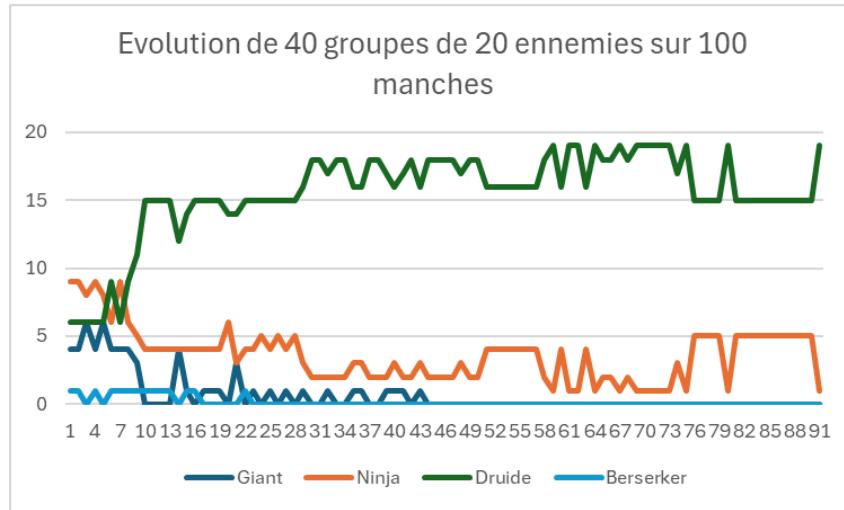
Ce graphique représente une exécution du programme sur un labyrinthe où les ninja (Fuyard) avait la possibilité de passer par un chemin sans défenses et donc de tout le temps arriver à la fin du labyrinthe. L'axe des ordonnées représente le nombre d'ennemis en fonction de sa classe. On voit bien que le processus d'évolution maximise rapidement le nombre de ninja car c'est eux qui sont les plus forts dans ces conditions.

PS : pas de condition d'arrêt sur cette exécution, sinon nous aurions pu observer la fin de la partie au niveau de la 23 manche.



Le graphique ci-dessus représente une exécution du programme sur 50 manches avec cette fois l'obligation pour les ninja de passer par un moins une défense. On voit que le programme cherche vite à maximiser les ninja tout de même mais cette fois avec des healer. L'observation que nous pouvons faire et que les meilleurs

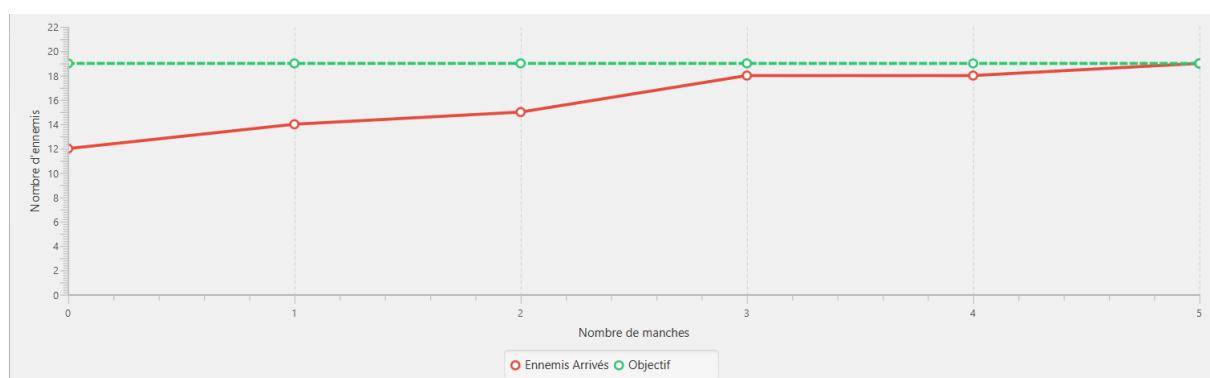
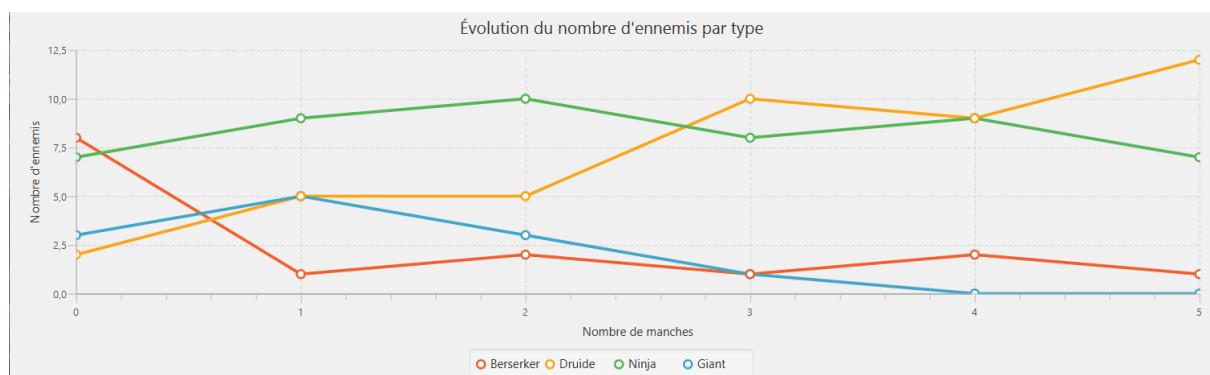
ennemis sont les ninja qui arrive a passé la tour sans se faire tuer grâce aux healer qui les soignent en même temps.



Ce graphique est une autre autre exécution du programme sur le même laby, cette fois sur 100 manches. On peut observer qu'au bout de la 91 manches les ennemis réussissent à gagner (ici il était condition de moins de 1 mort pour gagner) à la fin les ennemis gagnent avec une population importante de Druide (Healer) et une petite population du Ninja (Fuyard). La conclusion que nous pouvons en tirer est que la meilleur composition est composé d'un ninja, permettant d'avoir itinéraire ne passant que par une tour, et le reste des Druide car ils suivent le plus grand nombre d'ennemis, ici un Ninja, et heal les ennemis en continu, cela permet à tout le monde de passer la défenses sans morts grâce aux soins. Nous allons bien sûr optimiser tout ça pour la suite en rendant les druides un peu moins forts et tenter d'obtenir de différentes compositions en fonction du labyrinthe et des défenses.

Lors de ces dernières itérations, nous avons intégré des graphiques dans l'application qui nous permettent de voir en temps réel les résultats. Nous avons fait des simulations qui prouvent le bon fonctionnement de notre algorithme d'évolution.

Tout d'abord, nous avons fait un test sur un labyrinthe avec un chemin sûr que les fuyards, les ninjas, peuvent prendre sans se faire tuer. De ce fait, les ninja devraient être maximisés avec les druides qui suivent le plus gros groupe.



Comme attendu, les druides et les ninjas ont été maximisés et les ennemis ont réussi à passer le nombre nécessaire pour gagner assez rapidement.

Difficultés rencontrées

Nous avons eu de nombreuses difficultés au cours de notre projet, notamment liées à la mauvaise structure de notre code, ainsi qu'à la complexité de l'algorithme évolutionnaire.

En effet, au fil des itérations, nous avons ajouté nos différentes fonctionnalités sans nous soucier de la conception du code. Cela a particulièrement créé des dépendances indésirables entre les vues et les contrôleurs dans notre MVC, ainsi que des fonctions mal organisées au sein des différentes classes, ce qui a conduit notre modèle à être surchargé.

À titre d'exemple, lors de l'implémentation de l'attaque des entités, nous avons mis un compteur en temps réel dans le modèle, pour cadencer la vitesse d'attaque. Cela a engendré un bug, puisque le temps réel fonctionne avec l'interface graphique, qui fait avancer le jeu à une vitesse relativement lente, mais n'est pas adapté à la simulation qui va bien plus vite. Nous ne prenions pas en compte la vitesse du jeu dans notre vitesse d'attaque, ce qui a créé une incohérence entre interface graphique et simulation. Pour régler ce bug, il a suffi d'utiliser les pas de temps du moteur de jeu, ce qui donc permet à la vitesse des attaques des entités de rester cohérente quelque soit la vitesse du jeu. Ce bug a donc été créé par un ajout trop rapide de la fonctionnalité, sans réfléchir à son intégration dans le code existant.

Nous avons eu également de nombreux problèmes liés à l'évolution, qui n'a pas été fonctionnelle avant l'itération 4, dû à la complexité de l'algorithme, que nous avions mal saisi jusque-là.

Planning

Itération 1

Durant l'itération 1, sous les conseils de notre tuteur Vincent Thomas, nous avons réalisé plusieurs prototypes indépendants que nous devions fusionner par la suite. Cette méthode nous a permis de mettre en œuvre différents algorithmes afin d'évaluer leur efficacité et leur cohérence avec notre projet, avant de les fusionner ultérieurement. Un prototype a été réalisé pour chacun des algorithmes suivants : Algorithme évolutionnaire, A* et Steering Behaviours. Le moteur de jeu a de plus été récupéré du projet Zeldiablo (projet du Semestre 2) et adapté pour convenir à notre jeu avec une interface graphique telle qu'on l'avait imaginée. L'algorithme A* a été fusionné avec Steering Behaviours pour combiner le choix de chemin et un

déplacement fluide. Nous avons par ailleurs ajouté des types Ninja et Géants. Les Ninjas ont pour comportement “fuyard”, qui consiste en l’évitement de toutes les tours, et les Géants ont un comportement “normal”, c'est-à-dire suivre le chemin le plus court.

Itération 2

Pour cette itération, le but était d’abord de fusionner les prototypes précédemment réalisés, puis d’ajouter des fonctionnalités à notre jeu. Le prototype d’algorithme évolutionnaire a été fusionné au moteur de jeu et à l’interface graphique afin de pouvoir l’essayer directement dans le jeu. À ce stade, l’évolution ne fonctionnait pas à merveille, car notre démarche n’était pas forcément la bonne, on s’en est rendu compte plus tard. Des nouvelles fonctionnalités déjà pensées lors de l’étude préalable ont été ajoutées, notamment un panneau latéral de logs indiquant des informations au cours de la partie (quel ennemi meurt, quel ennemi évolue et de quelle manière). Une gestion des manches a été ajoutée afin de pouvoir passer d’une manche à l’autre avec une nouvelle population évoluée. Des sprites pour les différents ennemis, les défenses, les murs et les cases “chemin” ont été ajoutés. À cet instant, les défenses peuvent attaquer les ennemis qui entrent dans leur zone de portée. Les ennemis se déplacent différemment selon leur comportement (par exemple, les Ninjas priorisent un chemin sans défense tandis qu’un Géant ira au plus court sans se soucier des défenses). Nous avons ajouté deux nouveaux types, Druide et Berserker. Le Berserker va vers la tour la plus proche et la détruit. Le Druide, quant à lui, suit le plus grand groupe et soigne les ennemis dans une zone.

Itération 3

Pour cette itération, l’objectif principal était d’une part d’avoir une évolution fonctionnelle, et d’autre part d’ajouter de nouvelles fonctionnalités. L’évolution des ennemis a été modifiée afin que la reproduction se base sur les statistiques de naissance plutôt que sur celles de fin de manche. Cela a permis de corriger le principal dysfonctionnement de l’évolution, qui attribuait des stats très faibles aux nouveaux ennemis, puisque ceux-ci étaient créés à partir d’ennemis affaiblis, voire morts. Des améliorations ont également été apportées aux formules de score de l’évolution. Les ennemis peuvent désormais attaquer les défenses pour les détruire, ce qui entraîne un recalculation dynamique des chemins, et donc de s’adapter lorsque l’environnement change. Un refactoring du code des entités a été effectué, car avec l’ajout de l’attaque des ennemis, de nombreuses fonctionnalités étaient similaires entre ennemis et défenses. Nous avons également ajouté un comportement d’évitement des obstacles dans Steering Behaviours pour corriger des bugs où les ennemis se coinceraient dans un mur lors d’un virage. Enfin, nous avons implémenté la possibilité de lancer le jeu sans interface graphique, en “simulation”,

pour pouvoir faire des tests rapidement, notamment pour l'évolution, puisque la simulation peut effectuer de nombreuses manches très rapidement, contrairement à l'interface graphique.

Itération 4

Lors de cette itération, comme l'évolution n'était toujours pas fonctionnelle, nous nous sommes concentrés dessus pour être certain que l'évolution soit validée à la fin de cette itération. En parallèle, nous avons également implémenté un déplacement des ennemis effectués uniquement avec Steering behaviors, sans chemin calculé par A*. Nous avons dû pour cela améliorer la gestion des obstacles dans Steering behaviors, car celle implémentée au cours de l'itération trois comportait plusieurs bugs. Nous avons introduit une évolution par groupes afin d'optimiser la composition des vagues d'ennemis en fonction des performances des manches précédentes. Nous avons par ailleurs mis en place une évolution spécifique pour un agent seul, qui a pour but d'étudier l'évolution des statistiques de cet ennemi. Il est maintenant possible de générer des graphiques à partir des résultats des évolutions, qui sont stockés dans un fichier au format csv. Enfin, nous avons refactor une partie du code, notamment de la classe ModeleLabyrinthe, qui devenait trop grande et peu lisible.

Itération 5

Durant l'itération 5, nous avons ajouté des nouveaux affichages (mode simple, refonte des logs, ajout de sprites lors des ennemis et défense attaqués) pour une meilleure compréhension. Nous avons également ajouté un graphique affichant la vie des ennemis qui peut être suivi en direct et un qui affiche la composition des groupes d'ennemis générés à chaque fin de manche. La fonction de score n'étant toujours pas cohérente, nous l'avons donc modifiée. Nous avons ajouté la possibilité de modifier la vitesse du jeu en le ralentissant, l'accélérant, ou en faisant pause. Cela nous permet de pouvoir mieux comprendre ce qu'il se passe en ralentissant ou en mettant en pause tout en passant les moments répétitifs en accélérant le jeu. Le nombre de labyrinthes étant limité, nous avons ajouté la possibilité de charger un labyrinthe en format .txt. Les ennemis arrivant trop vite lorsqu'ils se déplacent uniquement avec Steering behaviors, nous avons ajouté un arrival behavior qui ralentit l'ennemi à proximité de l'arrivée. En effet, en arrivant trop vite, les ennemis dépassaient l'arrivée et tournaient autour. Arrival behavior est une force opposée à la direction de l'ennemi qui grandit à l'approche de l'arrivée. Enfin, nous avons fait de la résolution de bug et du refactoring.

Itération 6

Pendant cette itération, nous avons commencé l'évolution avec Steering behaviors qui consiste à faire évoluer les points de passage d'un ennemi pour qu'il prenne le meilleur chemin. Nous avons ajouté une fenêtre d'aide qui contient l'explication des différents ennemis, leurs comportements, leurs sprites ainsi que celle des défenses. Une fenêtre contenant les graphiques de l'itération précédente a été également créée. Pour que l'utilisateur ne soit pas perdu lors de l'évolution des ennemis, nous avons ajouté un pop-up qui montre l'évolution des ennemis. Nous avons aussi fait des configurations qui permettent à l'utilisateur d'avoir des scenarii pré-faits. Dans le même temps, nous avons fait une refonte graphique de la fenêtre de choix de paramètres. Enfin, nous avons refactorisé la classe MoteurJeu qui nous a permis de résoudre de nombreux bugs.

Itération 7

Lors de la dernière itération, nous avons avancé sur l'évolution avec Steering behaviors. Nous avons par exemple implémenté cette évolution dans une interface graphique pour pouvoir mieux voir les solutions que l'évolution pourrait nous proposer. Nous avons ajouté un nouveau graphique qui affiche le nombre d'ennemis passés par manche ainsi que l'objectif. Cela nous permet de pouvoir voir l'évolution du nombre d'ennemis qui passent par rapport à la composition qui est un graphique réalisé dans une itération précédente. Lorsqu'une partie était gagnée ou perdue, nous étions obligés de relancer le programme. Nous avons donc décidé d'ajouter un écran de fin (victoire ou défaite) avec la possibilité de recommencer. Dans le mode simple réalisé itération 5, les tours avaient pas de sprite associé, ce que nous avons fait lors de cette itération.

Conclusion

En conclusion, à l'issue de ce projet, nous avons un projet permettant de lancer une partie avec ou sans interface graphique, en choisissant le labyrinthe, le nombre d'ennemis, le nombre de manches, le nombre d'ennemis qui doivent passer pour gagner et si A* est utilisé ou non. Les ennemis évoluent à chaque manche, pour tenter d'atteindre l'objectif fixé au lancement de la partie, qui s'arrête lorsque les ennemis atteignent cet objectif ou que le nombre de manches maximum est atteint. Enfin, nous pouvons choisir entre une évolution sur les groupes d'ennemis, ou une évolution sur le mouvement des ennemis.

Nous avons donc une application fonctionnelle, qui possède bien plus de fonctionnalités que nous ne l'avions prévue lors de l'étude préalable. Ces fonctionnalités supplémentaires arrivent aux prix d'une mauvaise conception, puisque nous n'avons pas assez pris en compte la conception globale du code avant l'ajout de celles-ci, ce qui cause de nombreux bugs.

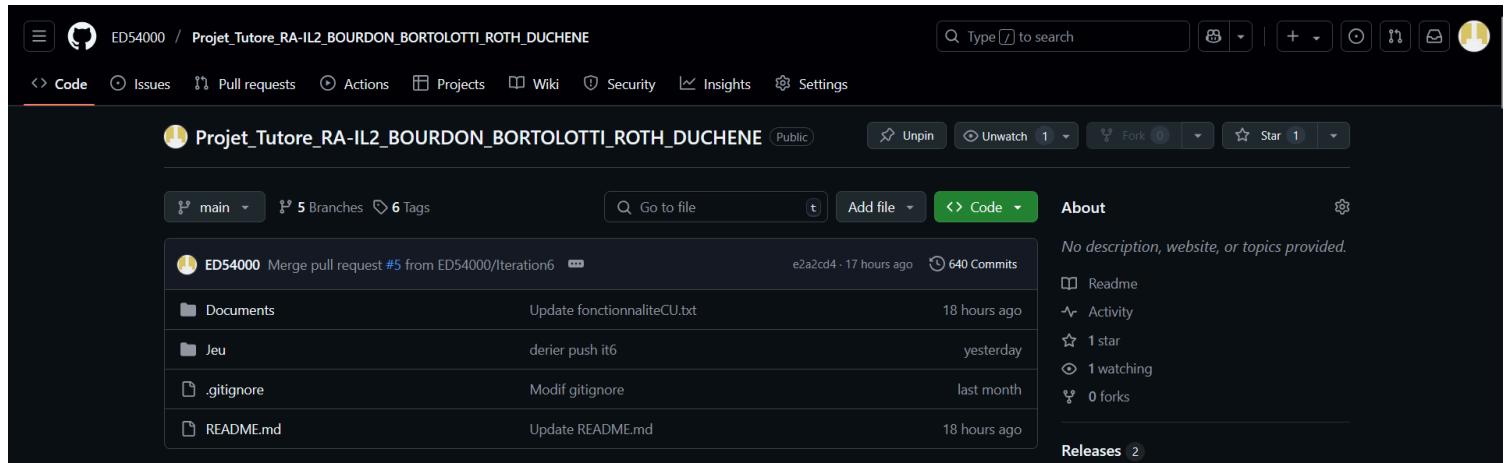
Ce fut donc un projet très enrichissant, puisqu'il nous a permis de découvrir de nouveaux algorithmes, notamment l'algorithme d'évolution, central dans ce projet. Il nous a également permis de nous améliorer énormément sur les aspects liés à la gestion de projet, en particulier la répartition des tâches, ainsi que la planification d'un projet sur une longue durée.

Nous ne pensons pas que notre projet puisse être repris par un autre groupe d'étudiants, en raison de sa mauvaise structure.

Annexe

Installer depuis GitHub

Sur GitHub, aller sur le dépôt d'Éloi (ED54000) et chercher le projet tutoré qui a pour nom [Projet_Tutore_RA-IL2_BOURDON_BORTOLOTTI_ROTH_DUCHENE](#).



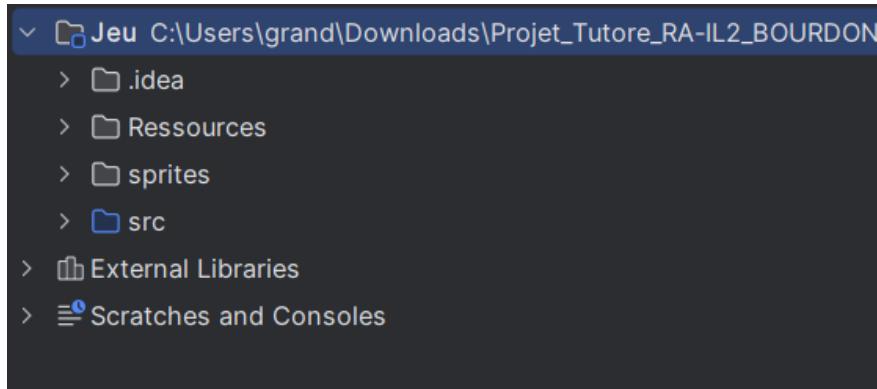
The screenshot shows a GitHub repository page. At the top, there are navigation links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The repository name is ED54000 / Projet_Tutore_RA-IL2_BOURDON_BORTOLOTTI_ROTH_DUCHENE. The repository is public. A search bar is at the top right. Below the header, there are buttons for Unpin, Unwatch (1), Fork (0), and Star (1). The main content area shows a list of files and commits. A commit by ED54000 is highlighted: "Merge pull request #5 from ED54000/Iteration6" (17 hours ago). The repository has 5 branches and 6 tags. On the right, there is an "About" section with a note: "No description, website, or topics provided." It also shows statistics: 1 star, 1 watching, 0 forks, and 2 releases. The "Code" dropdown menu is open, showing options like "Code", "Download ZIP", "Raw", "Copy", "Share", and "Edit on GitHub".

Une fois sur cette page, appuyer sur le menu déroulant Code et Download ZIP. Lorsque le ZIP est téléchargé, il suffit d'extraire le dossier pour que le projet soit téléchargé et prêt à être lancé. Le dossier contient trois parties : le README, le dossier Document qui contient les rapports et soutenances de fin d'itérations et le dossier Jeu qui est le projet en lui-même.

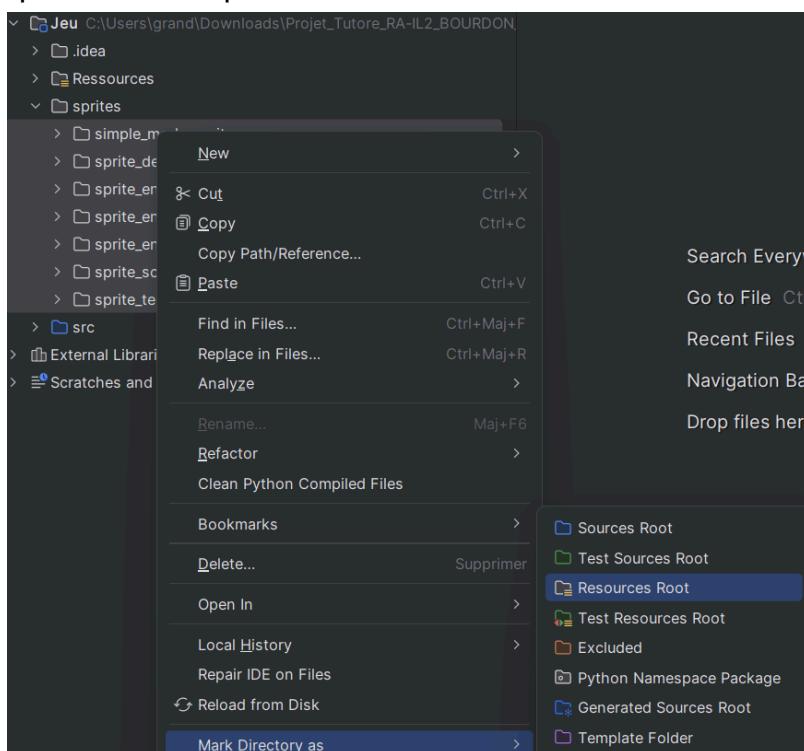
Lancer le projet

Avant de lancer le projet, il faut que les prérequis soient remplis. Le projet nécessite d'avoir Java 21 et JavaFX. Il faudrait de préférence ouvrir le projet avec IntelliJ IDEA qui sera utilisé lors de cette présentation.

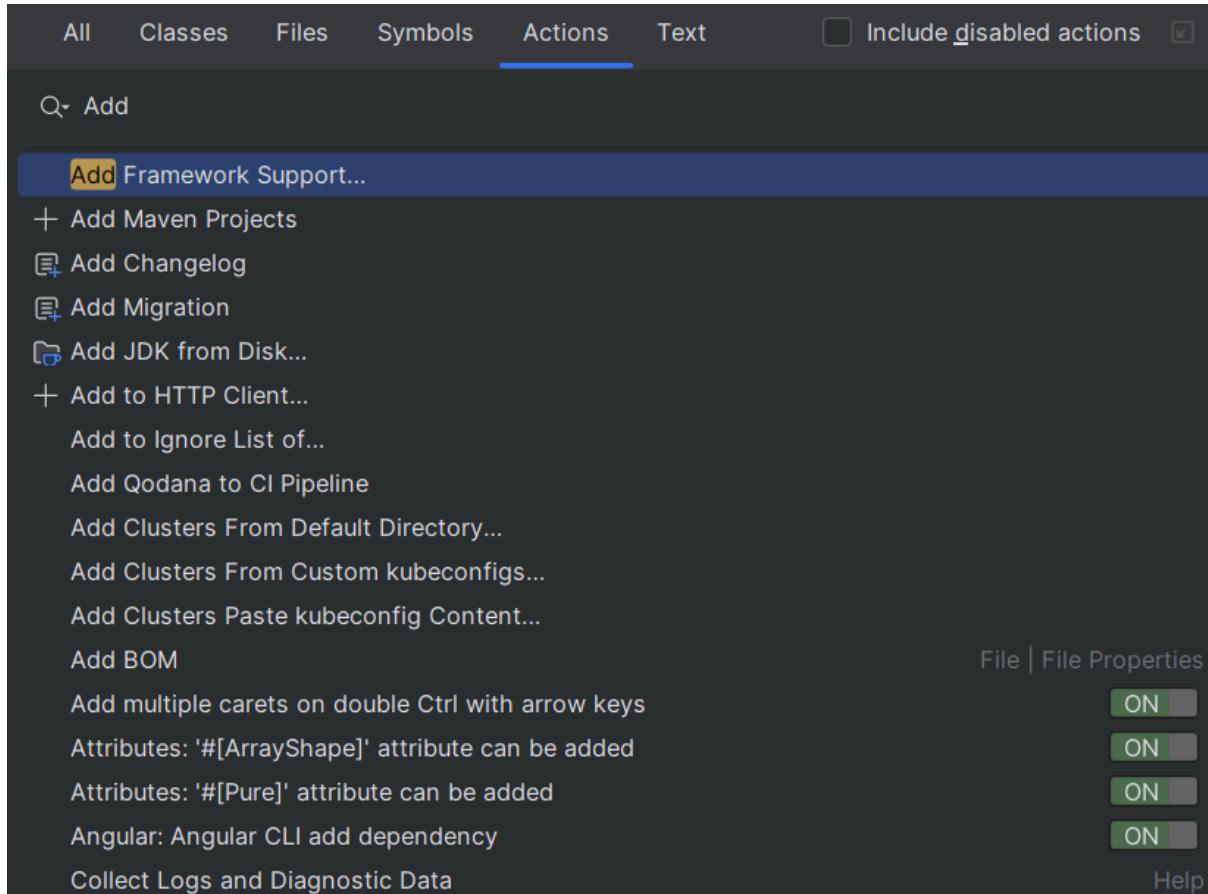
Ouvrez avec IntelliJ le dossier Jeu pour que l'arborescence du projet IntelliJ soit comme l'image ci-dessous



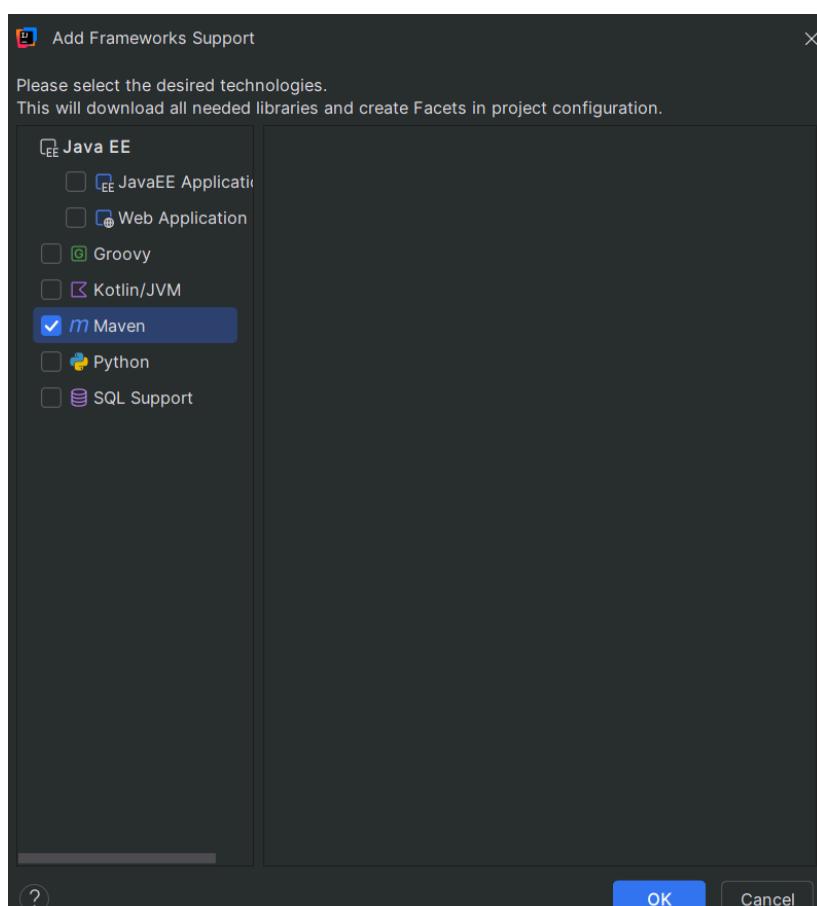
Le dossier Jeu qui contient trois sous-dossiers : Ressources, sprites et src. Pour que le projet fonctionne, il faut marquer le dossier Ressources ainsi que le contenu de sprites en tant que ressources Root.



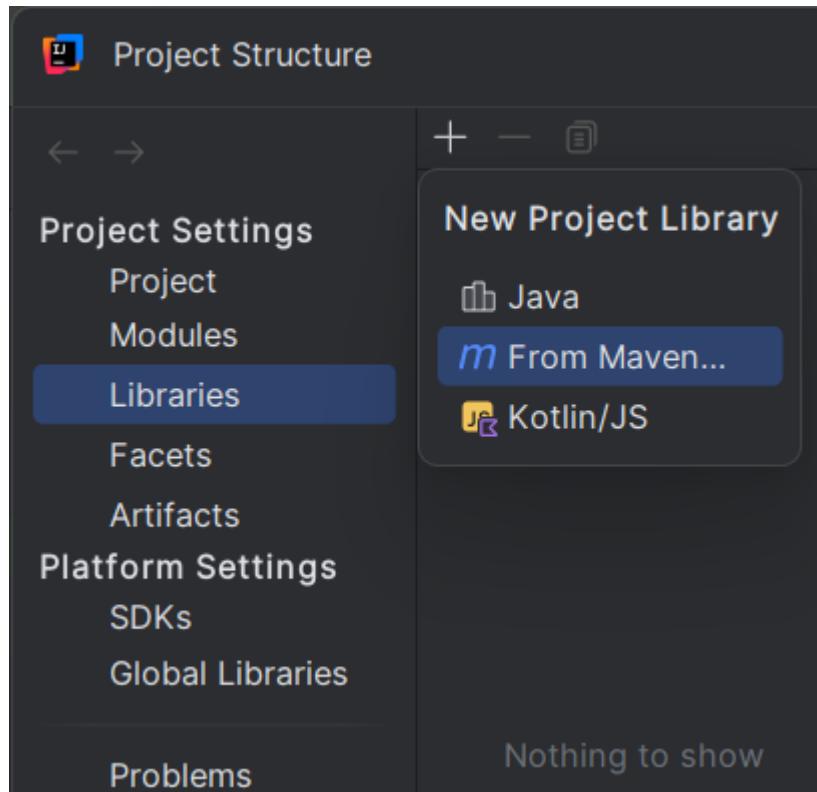
Ensute, il faut ajouter Maven au projet. Pour ce faire, cliquez sur le dossier Jeu et faites CTRL-SHIFT-A pour ouvrir cette fenêtre où il faut chercher Add framework support.



Sélectionnez Maven puis OK et cela ajoutera un pom.xml au projet.



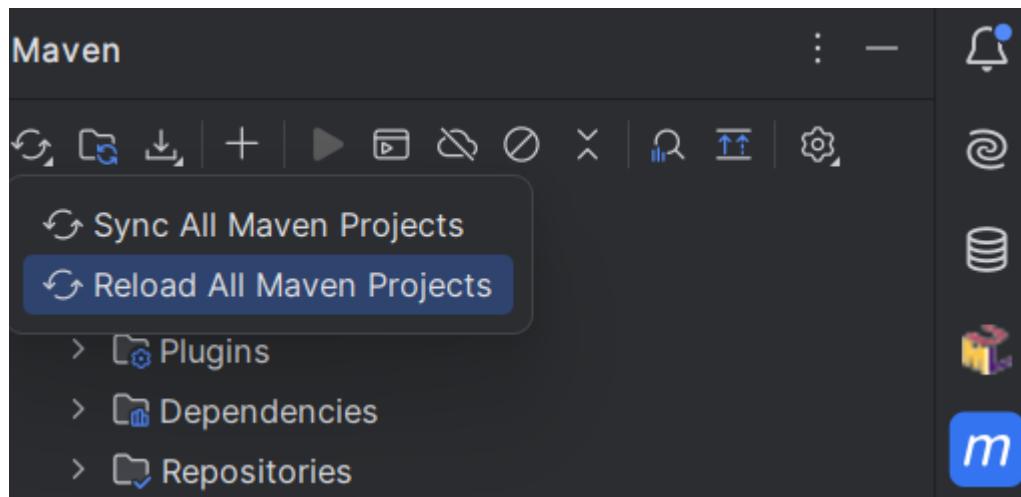
Il faut donc ajouter les dépendances JavaFx. Pour ce faire, ouvrez Projet structure et allez dans l'onglet Librairies. Cliquez sur le + et choisissez From Maven.



Une fenêtre s'ouvre et ajoutez org.openjfx:javafx-controls:21 ainsi que org.openjfx:javafx-base:21.

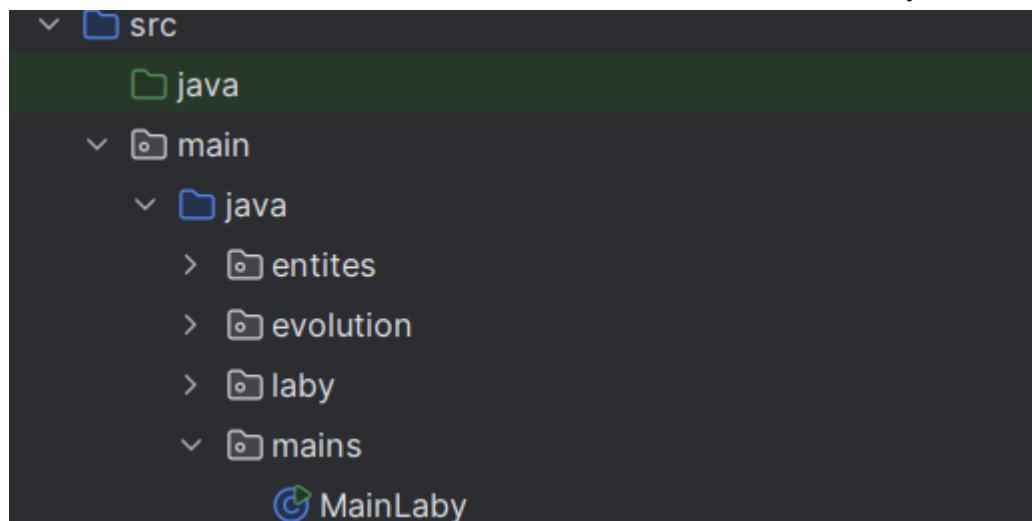
Si cela ne fonctionne pas, veuillez ajouter les lignes suivantes dans le pom.xml et relancer le Maven

```
<dependencies>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>21</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>21</version>
    </dependency>
</dependencies>
```

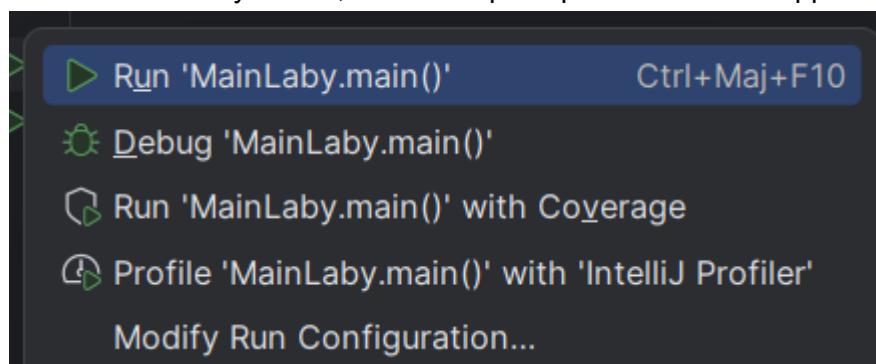


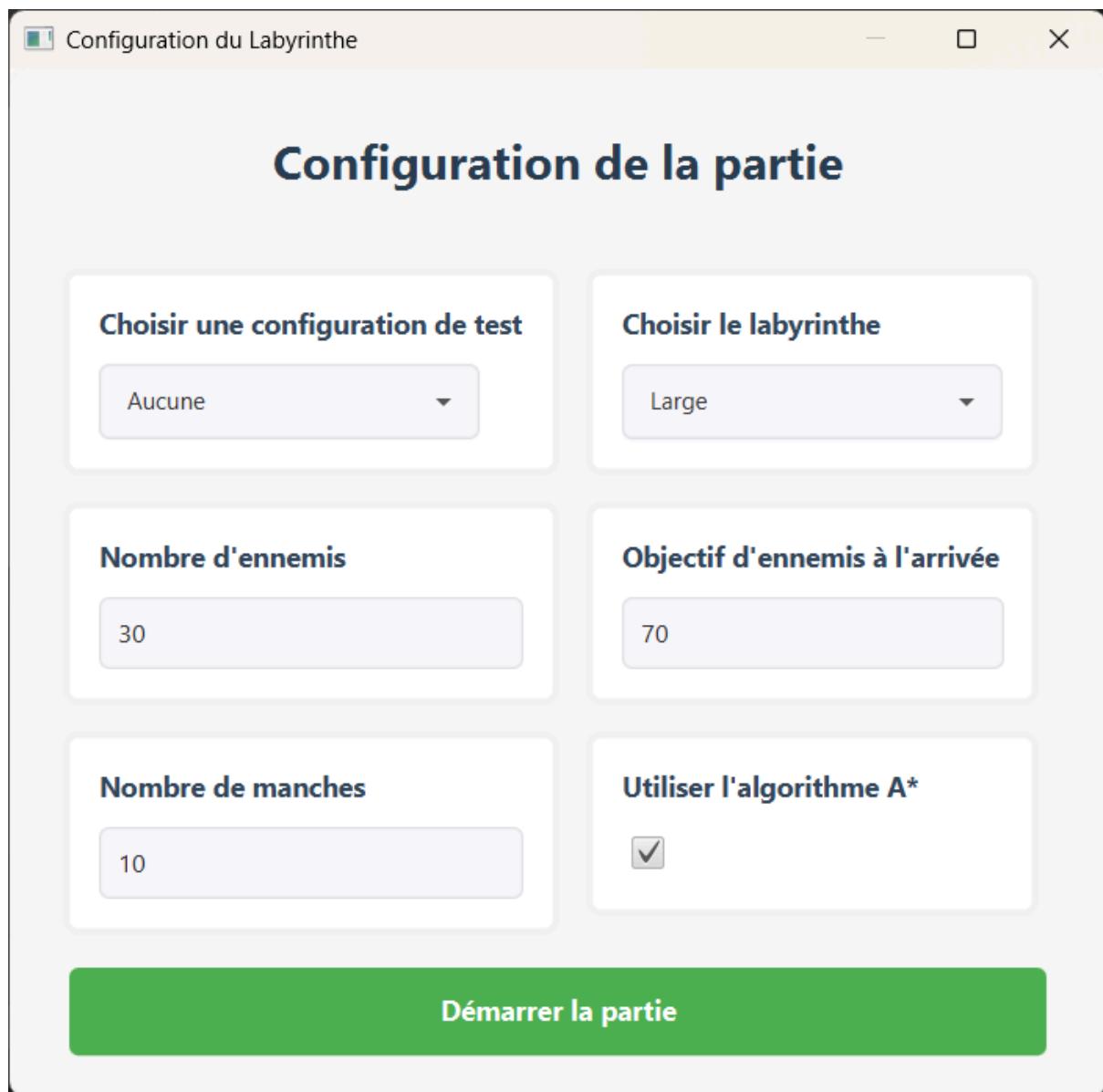
Une fois les dépendances ajoutées, le projet peut maintenant se lancer sans problème.

Pour ce faire, il suffit d'aller dans le dossier mains et d'ouvrir MainLaby.



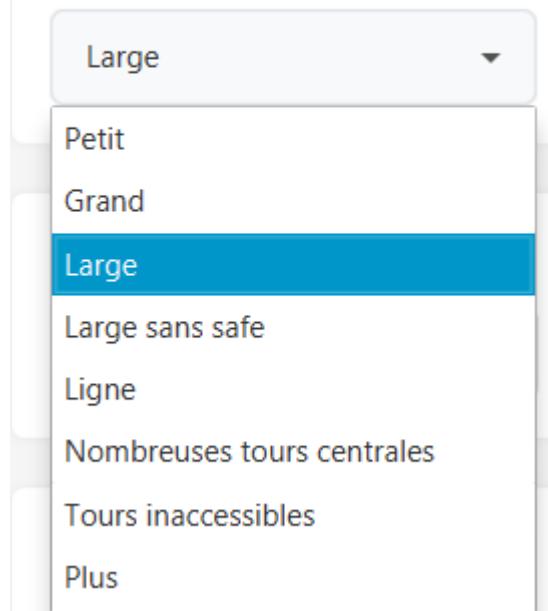
Une fois MainLaby ouvert, il ne reste plus qu'à le lancer et l'application se lance.





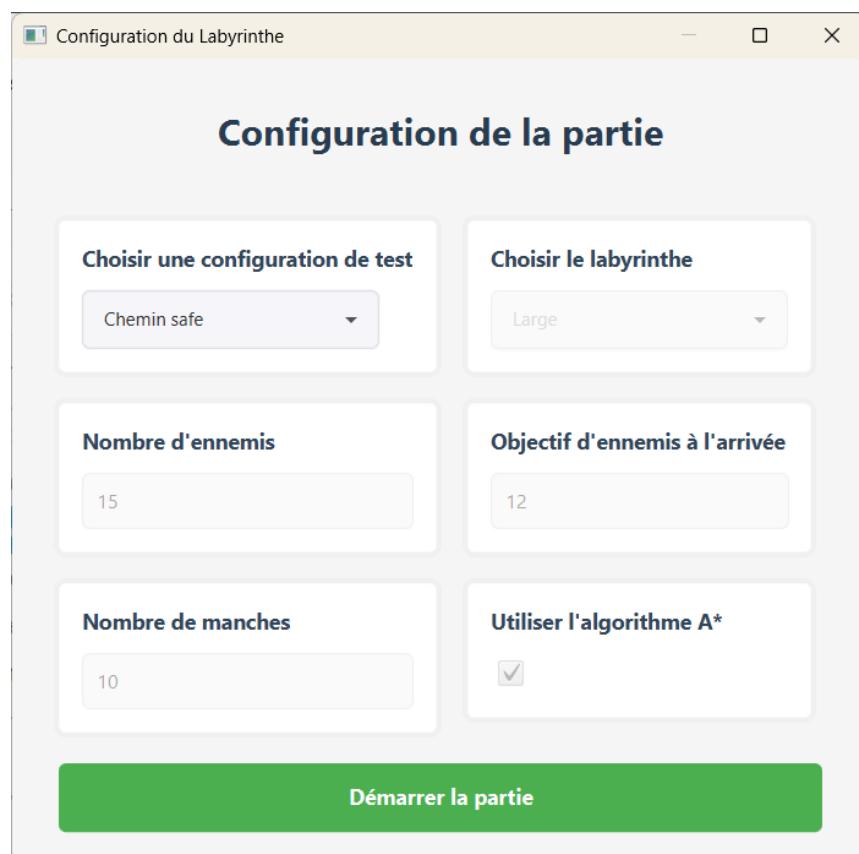
L'image ci-dessus est le panneau de configuration qui s'ouvre à chaque nouvelle partie. Vous pouvez choisir un labyrinthe déjà existant, ou en ajouter un nouveau en choisissant l'option "Plus". Attention, la fenêtre pour choisir le labyrinthe si "Plus" est sélectionné ne s'ouvre que lorsque vous démarrez la partie.

Choisir le labyrinthe

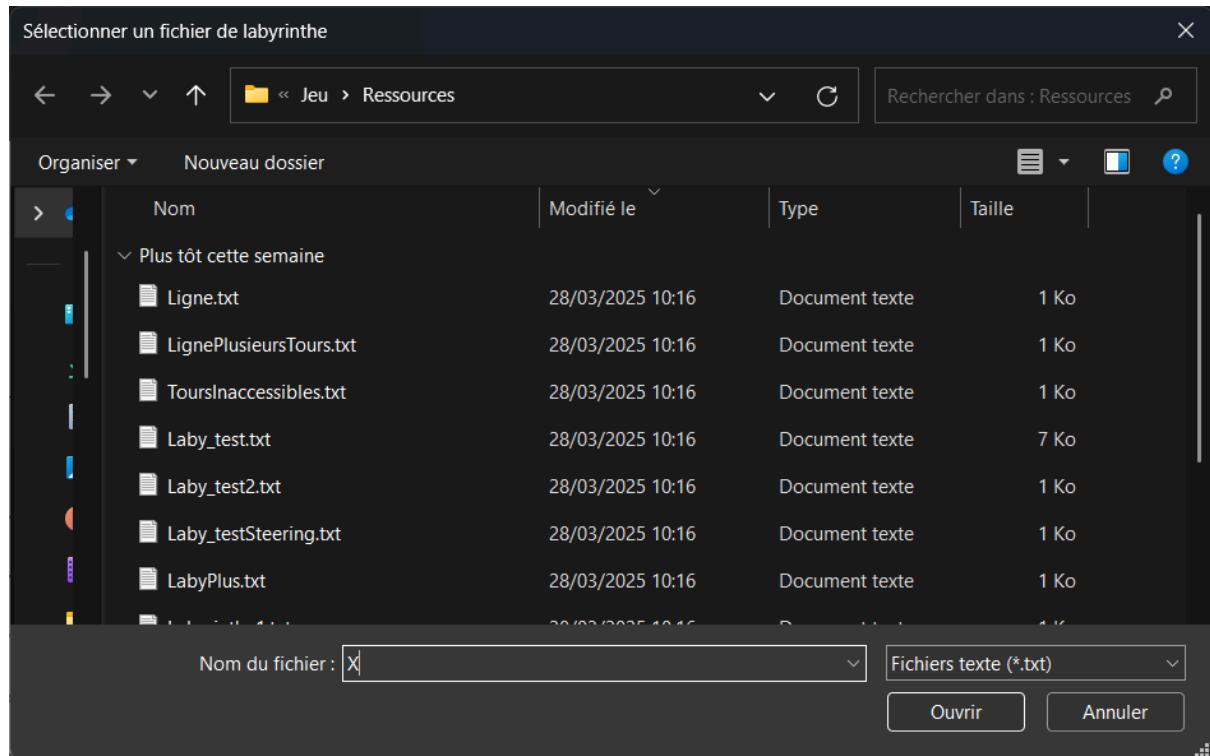


Ensuite, vous pouvez choisir le nombre d'ennemis, le nombre d'ennemis qui doit franchir l'arrivée pour gagner, le nombre de manches que les ennemis ont pour y parvenir et s'ils utilisent l'algorithme de choix de chemin A* ou s'ils se déplacent avec Steering behaviours.

Vous pouvez également choisir des configurations choisies par nos soins. Lorsque vous choisissez cela, le reste des paramètres ne sont pas modifiables et donc sont grisés.



Une fois tous les paramètres de la partie choisie, appuyez sur Démarrer la partie. Si vous avez mis "Plus" , une nouvelle fenêtre s'ouvre pour choisir le labyrinthe. Le fichier doit respecter certaines conditions qui seront décrites plus bas.



Le jeu se lance lorsque vous avez choisi un labyrinthe



Vous pourrez alors observer **l'évolution des groupes d'ennemis**.

Si vous souhaitez observer **l'évolution du mouvement**, il faudra changer de branche git et aller sur la branche "**Iteration7_steering**". Les procédés pour lancer la partie sont similaires sur cette branche.