Tower defense et Machine Learning
Rapport de fin de semestre

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

Tuteur : Vincent THOMAS

Année : 2024/2025

Table des matières :

Introduction

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.

Analyse

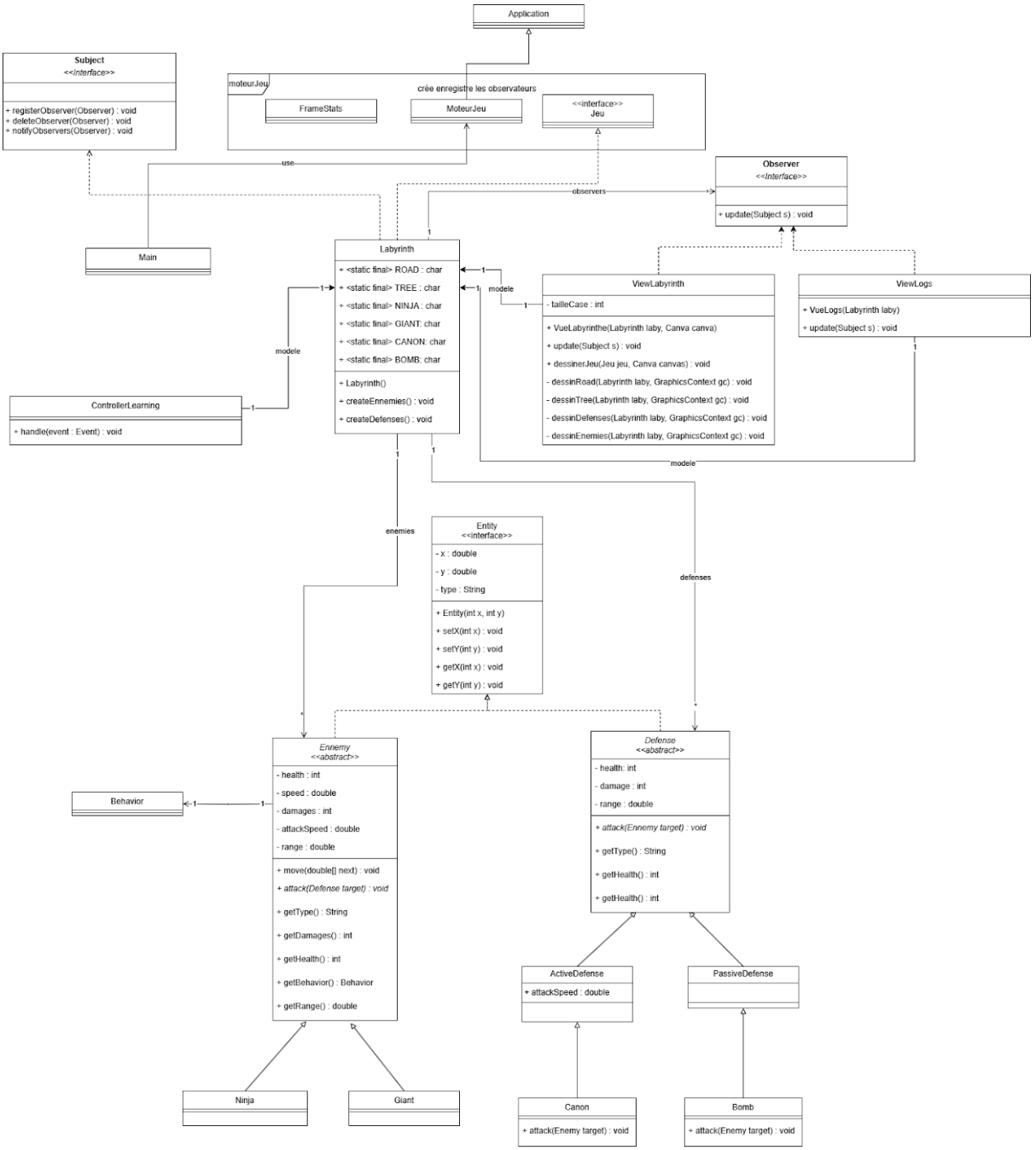
Projet initial

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

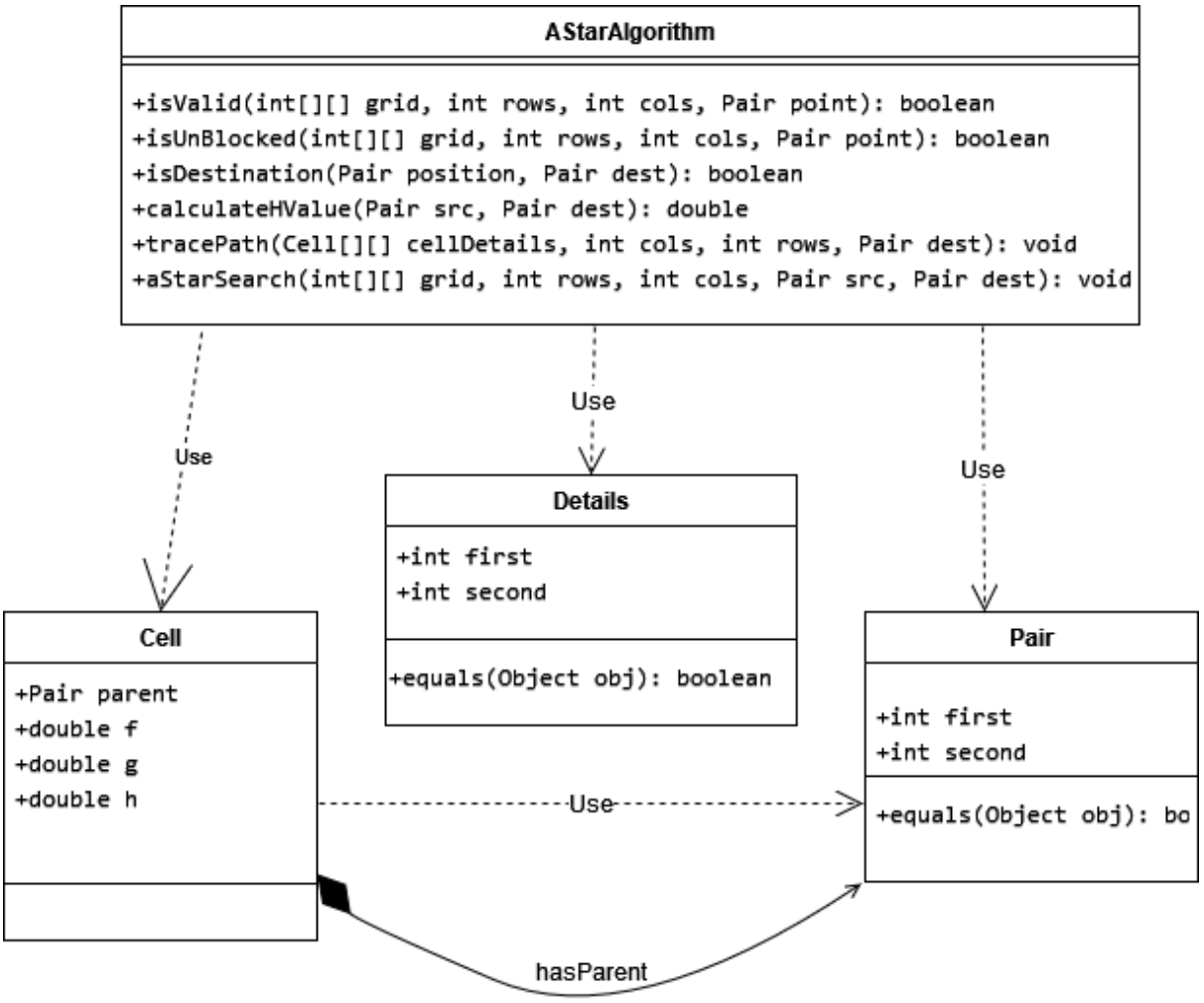
viable qui en résulte permet de faire tourner le jeu en créant un labyrinthe composé d'ennemis et de défenses, configuré via une interface utilisateur au lancement du programme, et représenté graphiquement.

Diagramme de classe correspondant :



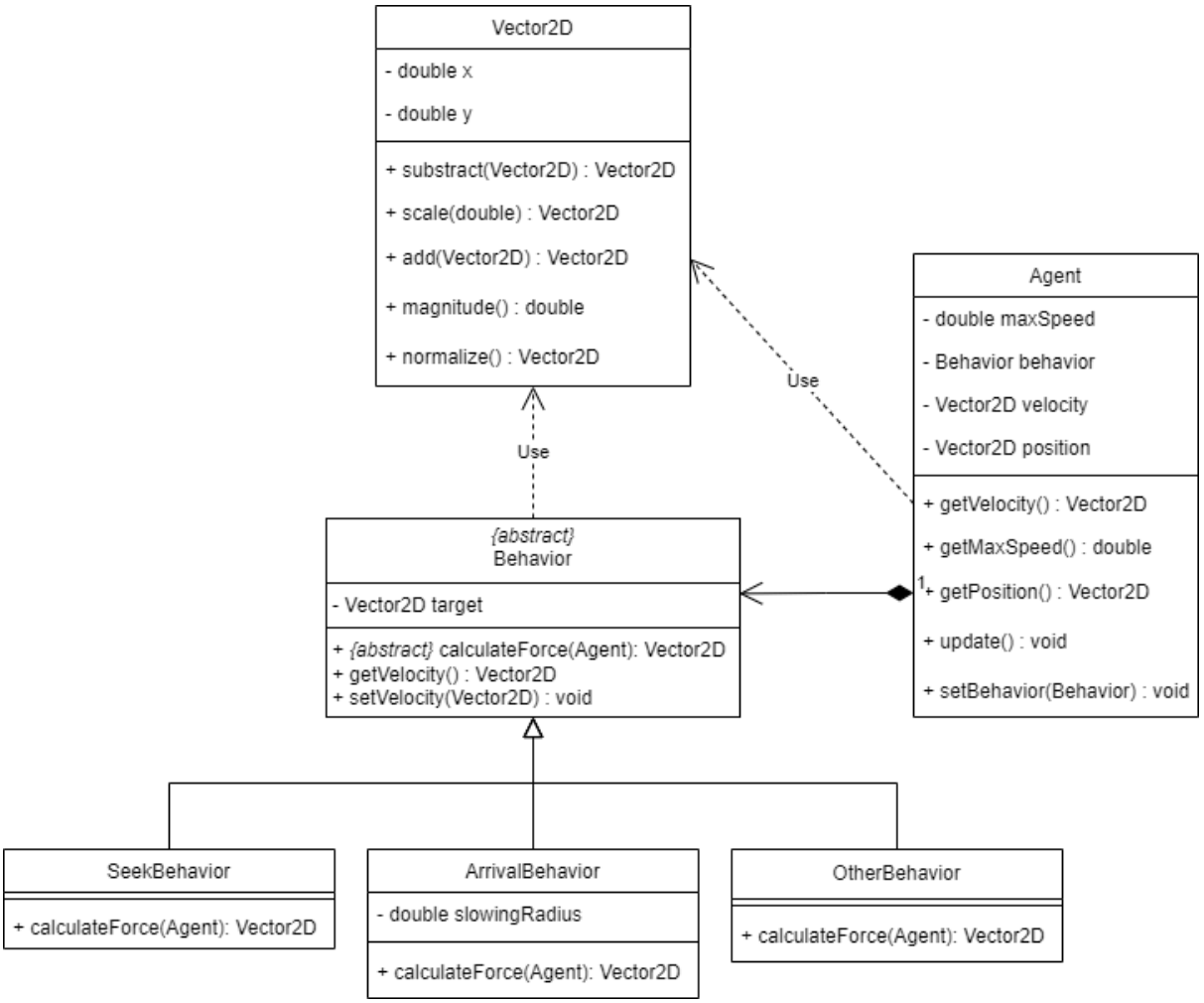
Ensuite nous avons le prototype de **l'algorithme de choix de chemin,** pour ce prototype, nous avons décidé d'utiliser l'algorithme A*. Il utilise une évaluation heuristique sur chaque nœud d'un graph pour estimer le meilleur chemin du nœud initial au nœud final, il visite ensuite les nœuds de ce chemin.

Diagramme de classe correspondant :



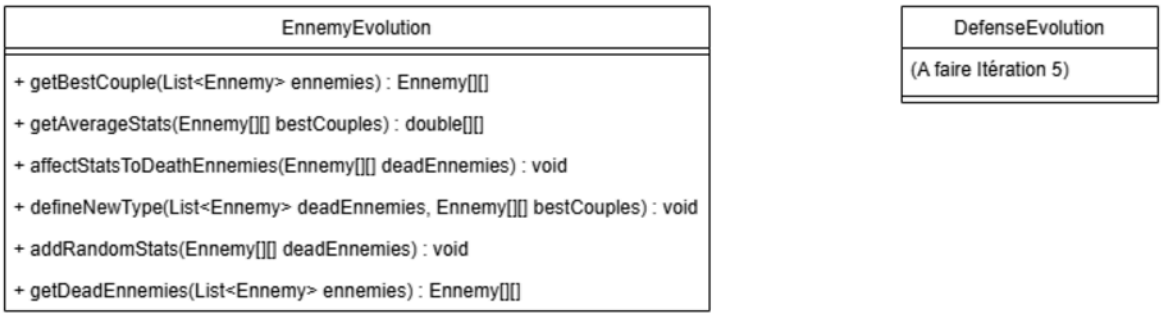
Nous avons également réalisé le prototype d'un **algorithme de déplacement,** représenté par un Steering Behavior, son comportement permet de simuler des mouvements réalistes en combinant plusieurs forces influençant la direction et la vitesse d'une entité. Le comportement de base inclut le suivi de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

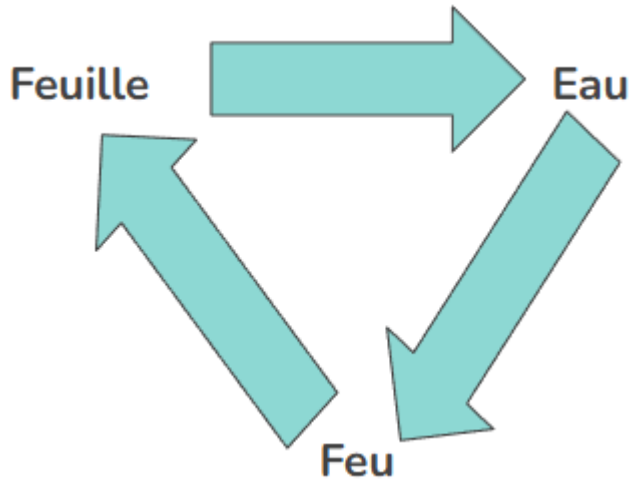


Maintenant que nous avons exposé les différents prototypes réalisé au début de notre projet, il est important de rappeler le **fonctionnement de nos ennemis**, ils possèdent plusieurs caractéristiques qui les définissent :

➕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.

Evolution de notre projet

Le fonctionnement de notre projet à 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**.

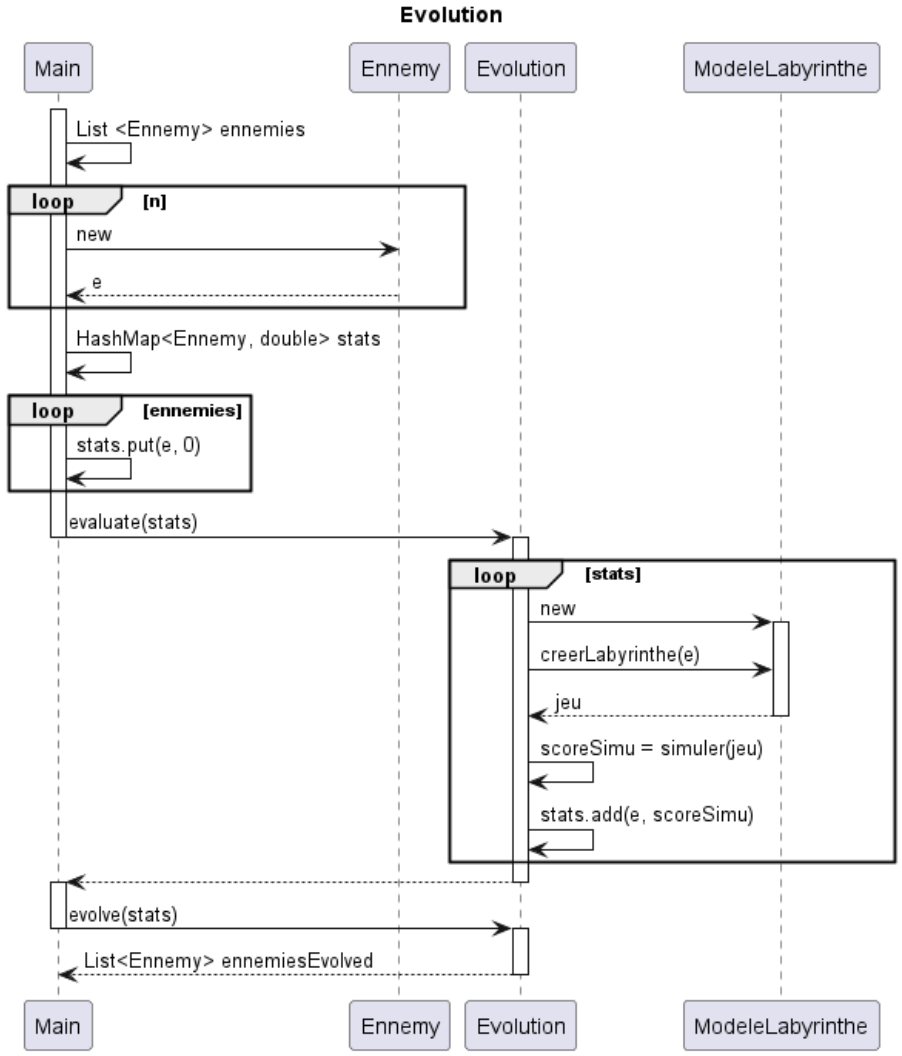
Diagramme de classe correspondant :

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 défini les contraintes relatives au choix du chemin de celui-ci par A*.

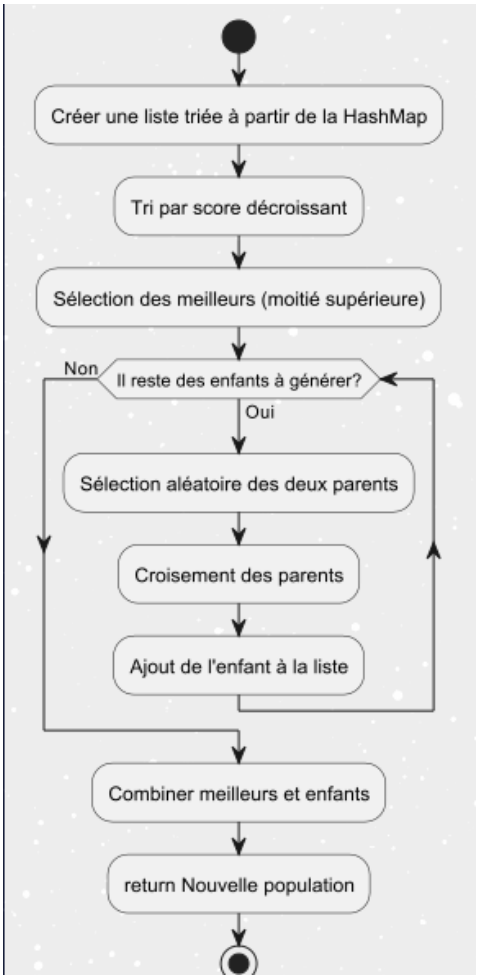
Nous utilisons tout de même **Steering Behavior** et avons décider de l'utiliser d'une autre approche, il est maintenant possible de le rendre indépendant de A* avec un déplacement de l'ennemi en fonction des forces exercées. L'objectif est par la suite de pouvoir apprendre les trajectoires des ennemis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

Notre approche par rapport à **l'évolution** des ennemis a elle aussi changé, nous avons repris l'évolution du début après avoir rencontré des problème pour l'intégrer à nos autres prototype, notre évolution est maintenant représenté par ce diagramme

de séquence :

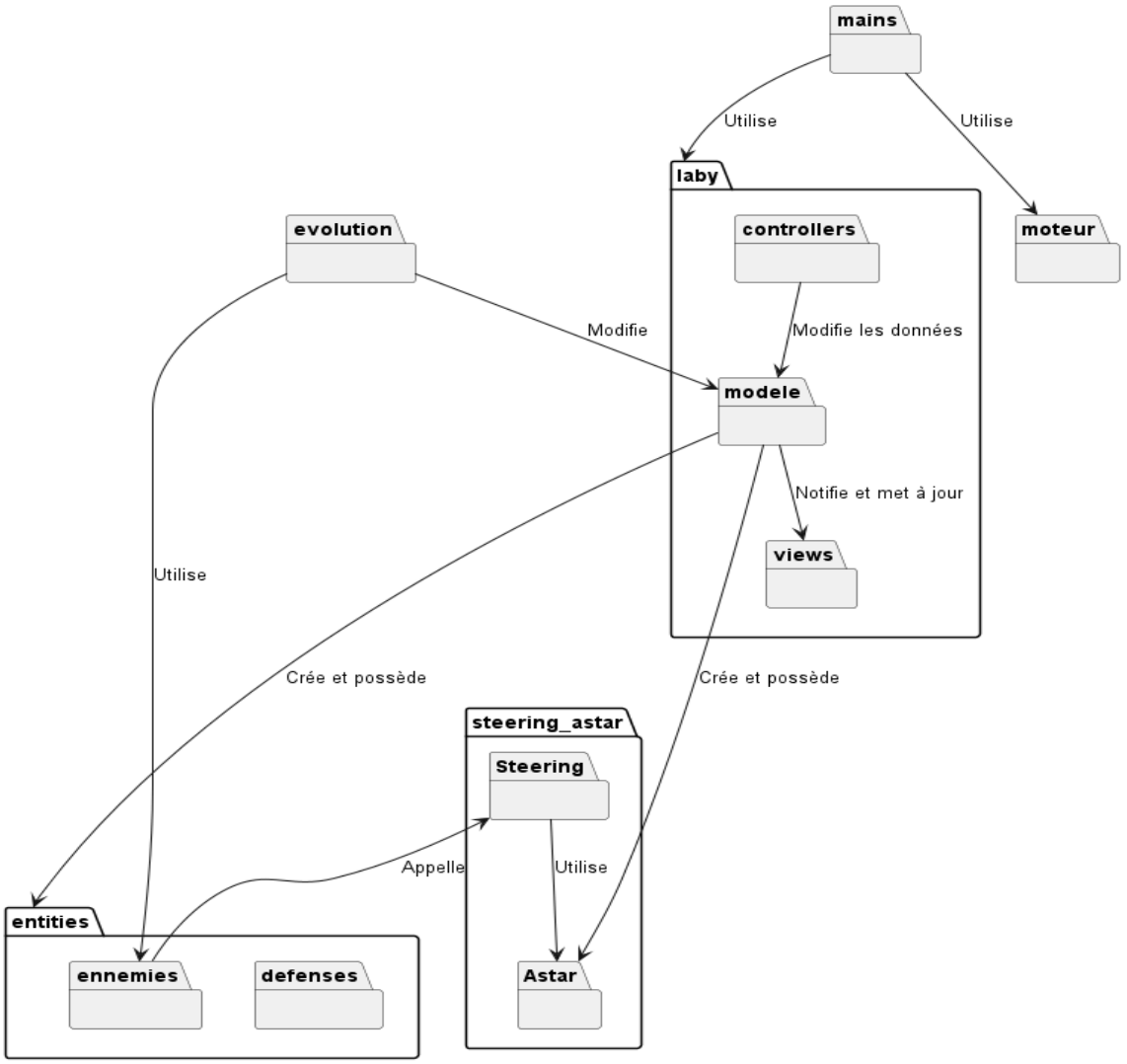

Evolution
Main
Ennemy
Evolution
ModeleLabyrinthe
List <Ennemy> ennemies
loop
[n]
new
e
HashMap<Ennemy, double> stats
loop
[ennemies]
stats.put(e, 0)
evaluate(stats)
loop
[stats]
new
creerLabyrinthe(e)
jeu
scoreSimu = simuler(jeu)
stats.add(e, scoreSimu)
evolve(stats)
List<Ennemy> ennemiesEvolved
Main
Ennemy
Evolution
ModeleLabyrinthe

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



Réalisation

Architecture logicielle

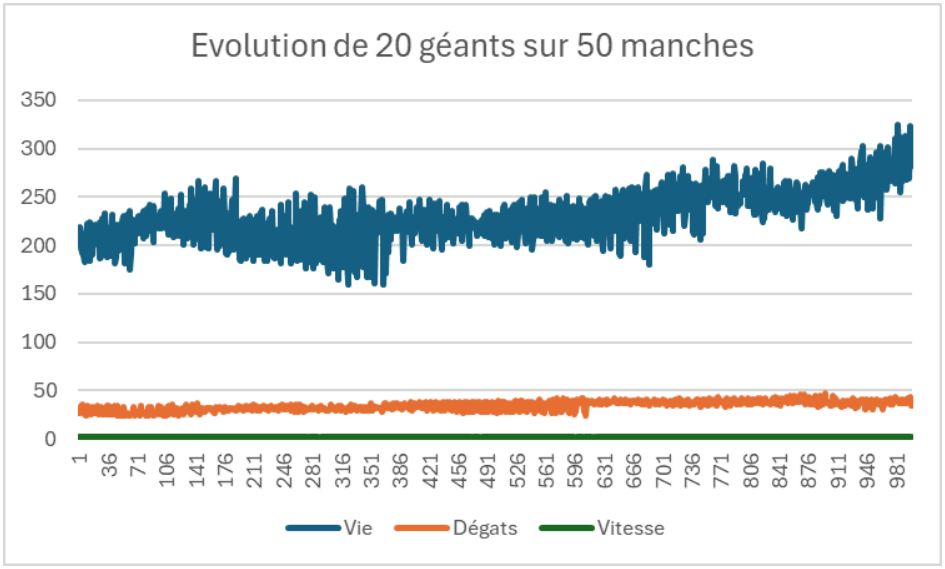


On peut voir sur le schéma ci-dessus la structure globale de notre application. On y retrouve notamment les prototypes réalisés durant l'itération 1 : le prototype du moteur de jeu correspond au bloc "laby", le prototype de l'évolution à l'élément "evolution", et enfin les prototype de A* et Steering 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*. L'évolution va utiliser les ennemis existants pour en créer de nouveaux, et va donc ensuite modifier la liste d'ennemis que possède le modèle en

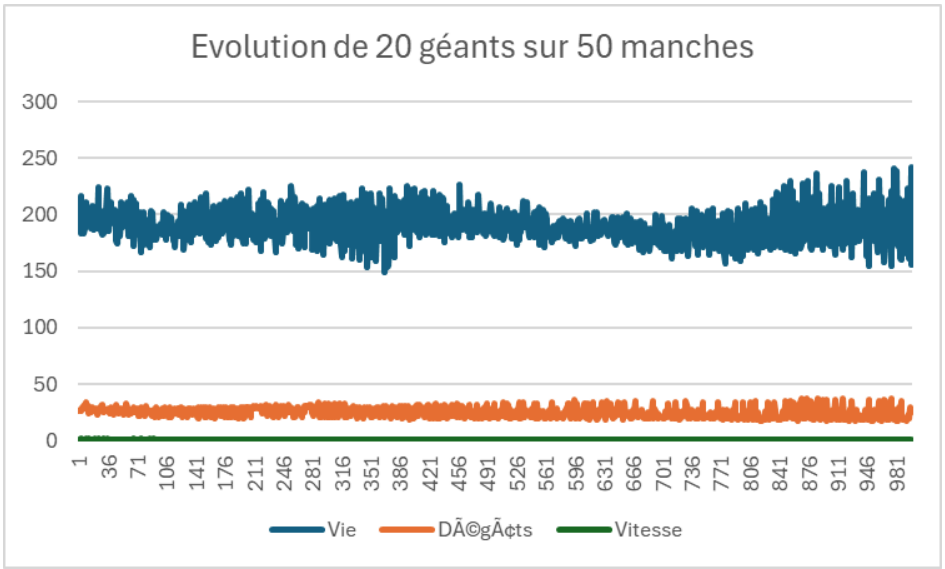
conséquence, en remplaçant une partie des ennemis existants avec les ennemis “évolués” nouvellement créés.

Test de validations

Évolution pour 1 individu

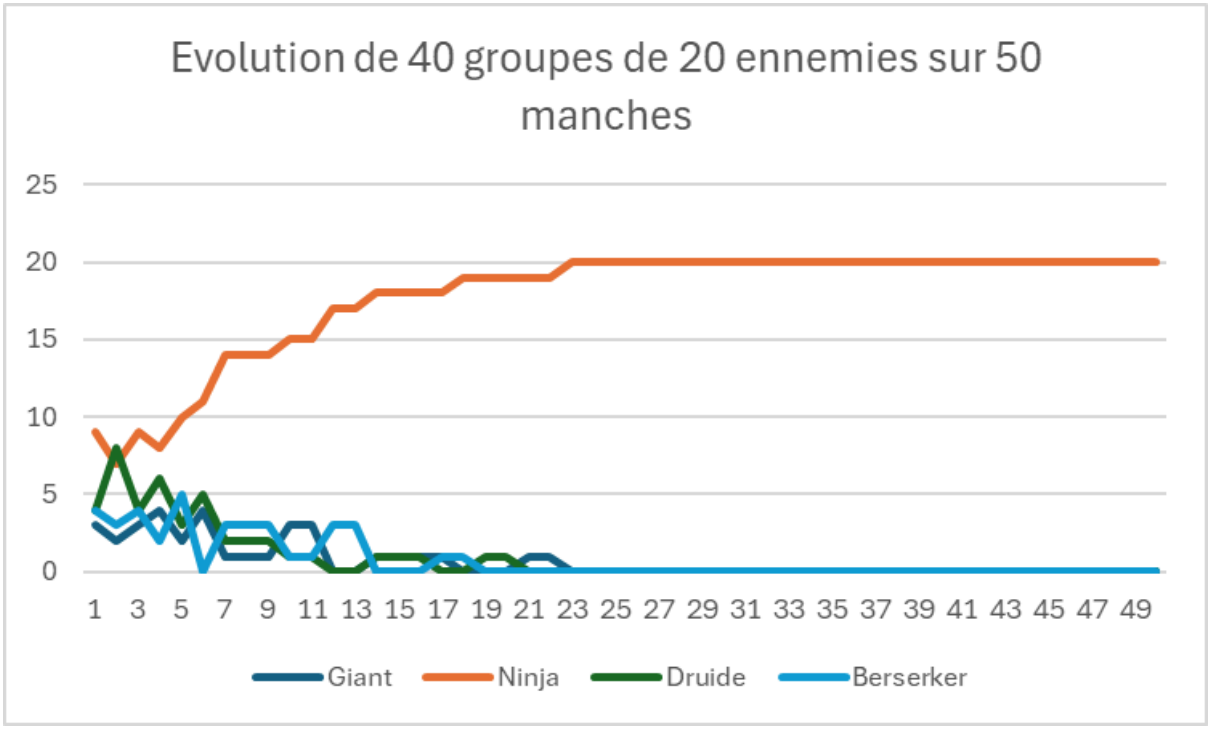


On voit que la courbe de vie diverge ici vers le haut et la vie augmente bien, L’axe des ordonnées représente la valeur réel de l’attribut de l’ennemi, et l'axe des abscisses représente lui le nombre de manche, ici il y en a bien 50 mais les valeurs incorrect sont dû une mauvaise création de graphique, 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 ennemis par la suite. Le fait d’afficher la liste est aussi la cause des grosses ondulations.

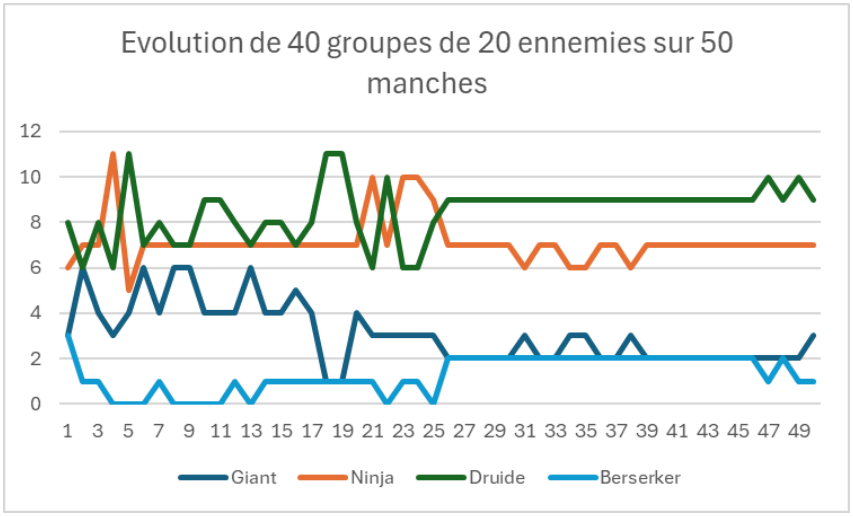


Ici l’exemple d’une autre exécution ou les statistiques ont stagné.

Évolution pour 1 groupe d'individus

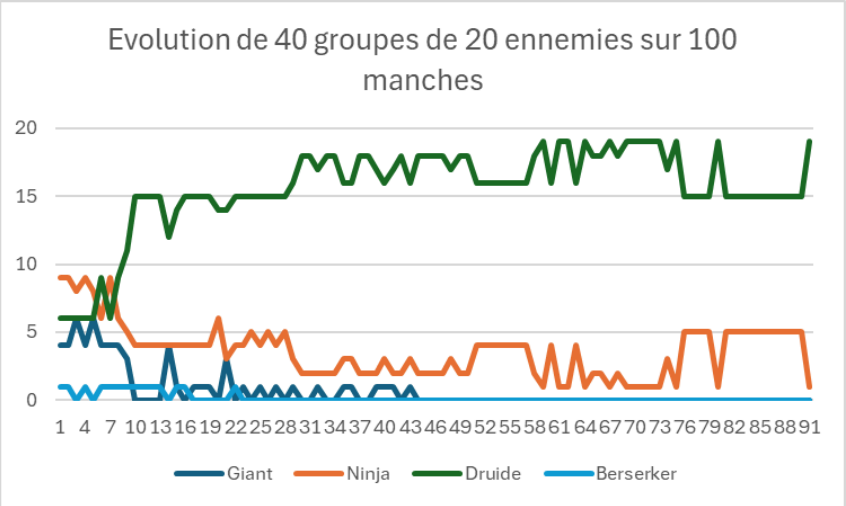


Ce graphique représente une exécution du programme sur un labyrinthe où les ninja (Fuyard) avait la possibilité de passer par un chemin sans défenses et donc de tout le temps arriver à la fin du labyrinthe. L'axe des ordonnées représente le nombre d'ennemis en fonction de sa classe. On voit bien que 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 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 continue, 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.

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 à notamment 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 à 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 suffit 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 à 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 par la suite. Un prototype a été réalisé pour chacun des algorithmes suivants : Algorithme évolutionnaire, A* et Steering Behaviours. Le moteur de jeu a également 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 recalcul 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 Behaviours, car celle implémentée au cours de l’itération 3 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 également 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.

Répartition du travail

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).

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'ennemi (évolution visant à faire évoluer la composition du groupe d'ennemi, en changeant les proportions de chaque type d'ennemi dans le groupe).

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.

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.

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.

Élément orignal

Tristan

L'un des éléments du projet dont je suis fier est la mise en place de la gestion des logs, en effet en mettant en place le modèle MVC j'ai créé une VueLogs qui permet à l'utilisateur de voir les informations de la partie en direct. Elle permet aussi à l'utilisateur de faire évoluer les ennemis à la fin d'une manche et d'afficher les informations relatives à l'évolution.

Marin

L'élément de notre projet dont je suis le plus fier est l'algorithme d'évolution. En effet, je travaille dessus depuis l'itération 2 et même si j'ai mis du temps à réellement saisir le concept de l'algorithme et à en produire un adapté à notre projet, je pense que c'est désormais un concept que j'ai acquis et ce fut très intéressant à développer. En fin de cette itération 4 donc, nous avons un algorithme d'évolution qui fonctionne, avec l'aide de Tristan.

Florian

L'élément dont je suis fier est l'implémentation de Steering behaviors, qui respecte un patron Strategy avec la classe Ennemy. La structure de l'implémentation est simple mais propre, ce que je trouve personnellement très satisfaisant.

Éloi

Pour ma part, l'élément original est la création de la grille de poids pour éviter les tours lors de l'utilisation de l'algorithme A*.

Objectifs à atteindre

Itération 5

Pour l'itération 5, nous avons prévu de :

- Ajouter l'évolution avec steering behavior, en définissant des points de passages dont les coordonnées vont évoluer
- Ajouter un arrival behavior (l'ennemi ralenti quand il arrive à proximité de l'arrivée), pour éviter les bugs où l'ennemi n'arrive pas à atteindre l'arrivée
- Pouvoir charger un labyrinthe depuis un fichier txt local
- Ajouter l'accélération du temps en version graphique, ainsi que la possibilité de mettre le jeu en pause
- Intégrer une génération de graphique en fin de partie

L'objectif principal de cette itération sera la mise en place de l'évolution sur Steering behaviors (dans le cas où A* n'est pas utilisé) par Marin et Florian. Nous avons aussi prévu quelques fonctionnalités secondaires, comme le fait d'ajouter un arrival behavior, par Éloi et Florian, ainsi que le fait de charger un labyrinthe depuis un fichier local, qui sera réalisé par Éloi. Enfin, Tristan implémentera la fonctionnalité de gestion du temps sur l'interface graphique ainsi que la génération de graphique en fin de partie.

Itération 6

Pour l'itération 5, nous avons prévu de réaliser une "course à l'armement", c'est-à-dire l'implémentation d'une évolution sur les défenses. Cela nous permettra d'observer de nouveaux comportements des ennemis, qui devront peut-être essayer "d'anticiper" les futures évolutions des défenses, et inversement.
Nous ne prévoyons pas plus de fonctionnalités pour cette itération, en anticipation d'éventuelles complications lors de l'itération 5, sur l'évolution du mouvement des ennemis, ainsi que de nouveaux bugs à fixer.

Itération 7

Comme les deux itérations précédentes comportent déjà chacune une fonctionnalité majeure (faisant intervenir l'évolution), nous avons choisi de ne pas ajouter de fonctionnalité importante en itération 7.
Cette itération sera consacrée à la finalisation du projet, donc à fixer les bugs restants des précédentes itérations, terminer les fonctionnalités non finies des itérations précédentes.
Nous prévoyons aussi d'améliorer les logs de l'interface graphique, qui sont actuellement limités et peu lisibles.

Conclusion

En conclusion, à l'issue de l'itération 4, 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 manche maximum est atteint.

Nous avons donc une application fonctionnelle, qui possède bien plus de fonctionnalités que nous ne l'avions prévues 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. Nous allons donc au cours des prochaines itérations essayer d'intégrer au mieux les futures fonctionnalités au code actuel, pour minimiser le nombre de bugs créés. Il faudra donc passer plus de temps sur la conception de ces nouvelles fonctionnalités, plutôt que sur leur implémentation.