Tower defense et Machine Learning

I) Définition du projet

→ Un tower defense, qu'est-ce ?

Le ***tower defense*** (souvent abrégée en **TD**) est un type de jeu vidéo où l'objectif est de défendre une zone contre des vagues successives d'ennemis se déplaçant suivant un itinéraire ou non, en construisant et en améliorant progressivement des tours défensives.

Et dans notre cas ?

Notre objectif dans ce projet est de se placer du côté des ennemis. On doit se concentrer principalement sur la manière dont les ennemis vont réagir, comment ils vont se déplacer, comment ils vont attaquer une défense après plusieurs essais, en essayant d'apprendre de leurs erreurs en utilisant de l'intelligence artificielle.

Sur quoi va-t-on se baser pour l'apprentissage des ennemis ?

On a plusieurs critères "de base" avec lesquels on peut se faire une idée, mais ce ne sont que des exemples, on pourra penser à d'autres critères plus tard pour l'apprentissage :

- Propriétés classiques (point de vies, vitesse de déplacement, dégâts)
- Un système de type "style Pokémon" avec des "contre types" (une défense type eau sur un ennemi type feu sera très efficace par exemple, et inversement, on peut imaginer ça avec un multiplicateur de dégâts en fonction des types par exemple).
- La forme de trajectoire des ennemis sur la carte (position, vitesse, accélération, point de passage) → Observer les tendances de déplacement, comment s'en servir ? Quels types de déplacement sont les plus efficaces ?
- La manière dont les ennemis prennent leurs décisions. Vont-ils prioriser une défense d'un type faible face au leur ? Vont-ils attaquer à plusieurs ? Vont-ils focus une même défense ou vont-ils chercher à s'améliorer même quand la manche précédente s'est bien passée pour eux ?
- La manière d'attaquer en groupe. Plusieurs types en une attaque ? Dans les premières manches ou plus tard dans la partie ? → Déterminer la manière la plus efficace d'attaquer en groupe.
- Les défenses seront fixes, comment les placer stratégiquement pour faire réagir les ennemis différemment ?

Que peut-on tester ?

On va devoir tester plusieurs algorithmes d'apprentissage différents.
On doit tester plusieurs "maps" différentes en positionnant les défenses de manière judicieuse, certaines avec des labyrinthes, afin de voir l'évolution des comportements des ennemis.
On va devoir tester les associations de types (types d'ennemi, type de défense) pour voir la réaction des ennemis (faire en sorte qu'ils ciblent les défenses de type faible par rapport à leur propre type).

Étudier la stratégie des ennemis, voir lequel ira en premier attaquer les défenses (le plus rapide ? celui qui a le plus de vie ?) en fonction de la map testée.

II) Étude des solutions existantes

1. Étude des moteurs de jeu possible

Différentes approches pour les moteurs de jeu :

- **Slick2D**
  Bibliothèque intégrant un moteur de jeu extrêmement simple d'utilisation avec une prise en main facile et une structure de code claire pour la gestion des boucles de jeu. On retrouve un rendu graphique performant, une gestion d'évènement simple pour les entrées clavier et souris, gestion de base de la physique et des collisions, intégration de Tiled qui est un éditeur de carte pour créer facilement des niveaux au cas où nous créons plusieurs labyrinthes. À noter que la bibliothèque Slick2D n'est plus activement maintenue.

- **JavaFX**
  Bibliothèque graphique qui permet de créer des interfaces utilisateur pour Java, cet outil nous est déjà familier étant donné que nous avons réalisé plusieurs projets à l'aide de JavaFX en cours tel que "Zeldiablo" ou encore une application de planifications de tâches similaires à "Trello". L'avantage principal est que nous connaissons déjà ce framework, mais celui-ci pourrait rapidement poser des problèmes et ralentir le cœur du développement si nous rencontrons des soucis liés à l'utilisation du moteur alors que ce n'est pas la partie sur laquelle nous devons passer le plus de temps.

- **Swing**
  bibliothèque graphique incluse dans Java, c'est l'outil standard de Java pour créer des interfaces, on retrouve donc une large documentation pour cet outil. L'intégration est donc facile et c'est parfait pour des jeux très simples voir statiques, mais les performances des interfaces liées à Swing peuvent vite être limitées, ce n'est donc pas optimisé pour un rendu graphique rapide, mais peut tout de même être utilisable dans le cadre de notre projet. On retrouve de plus beaucoup de projet simple créé en Swing ce qui fait encore une fois office d'une large documentation, exemple : https://members.loria.fr/vthomas/mediation/JV_IUT_2016/#ia

2. Étude de la représentation des comportements des ennemis possibles (Le déplacement des ennemis)

Nous allons ici comparer trois solutions :

- **Les “steering behaviours” :**

  https://www.red3d.com/cwr/steer/gdc99/

  Fonctionnement :

  Ces comportements permettent de simuler des mouvements réalistes en combinant plusieurs forces influençant la direction et la vitesse d'une entité. Les comportements de base incluent le suivi d'une cible, l'évitement d'obstacles, la fuite, le suivi de chemin, etc.

  Chaque comportement produit une force directionnelle, et la somme de ces forces est utilisée pour déterminer la trajectoire de l'entité. Les poids peuvent être ajustés pour prioriser certains comportements par rapport à d'autres.

  Formule du déplacement : $v_{t+1} = v_t + a_t \times dt$.

  avec v_t la vitesse a un instant t, a_t l’accélération et dt la durée.

  Pour implémenter un apprentissage via ce mode de déplacement, il faudra donc faire varier a_t.

  Avantages :

  - Peut donner des mouvements assez complexes avec un bon paramétrage.

  Inconvénients :

  - Les réglages manuels peuvent devenir compliqués puisqu’il faut gérer plusieurs paramètres en parallèle.
  - Un mauvais paramétrage peut donner des résultats très éloignés des résultats attendus.

- **Les points de passage**

  https://en.wikipedia.org/wiki/Waypoint

  Fonctionnement :

  Les entités se déplacent de point en point pour suivre une trajectoire déterminée. Les waypoints peuvent être disposés en ligne droite, en cercle,

ou en réseau plus complexe pour gérer différentes routes, intersections, ou destinations possibles.

Le passage d'un waypoint à un autre peut être régulé par divers critères, comme la distance à parcourir, les obstacles à éviter, ou des déclencheurs d'événements (par exemple, un waypoint qui déclenche un changement de comportement à l'approche d'une entité).

Avantages :

- Cette solution est très simple à paramétrer, puisque la position des points suffit.
- La simplicité de cette solution sera particulièrement utile pour tester les premiers algorithmes, car le comportement des ennemis sera parfaitement prévisible.

Inconvénients :

- La simplicité de cette solution deviendra un obstacle lorsque nous voudrons adopter des comportements d'ennemis plus complexes.

- **Les machines de Braitenberg**

https://en.wikipedia.org/wiki/Braitenberg_vehicle

Fonctionnement :

Les machines de Braitenberg sont des véhicules imaginaires dotés de capteurs et d'actionneurs, connectés de manière simple pour générer des comportements complexes. Par exemple, un véhicule avec deux roues et deux capteurs peut être programmé pour tourner vers une source de lumière si les capteurs détectent de la lumière. La connexion directe entre les capteurs et les actionneurs produit des comportements qui peuvent sembler sophistiqués, bien que le principe de base soit très simple.

Les machines de Braitenberg illustrent comment des comportements apparemment intelligents peuvent émerger de règles très simples.

Avantages :

- Permet des mouvements assez complexes en définissant quelques comportements simples.

Inconvénients :

- Les scénarios très complexes peuvent demander de définir beaucoup de règles pour obtenir un résultat satisfaisant.

3. Étude des algorithmes possibles (L'évolution des ennemis)

Nous allons comparer deux algorithmes d'amélioration des ennemis :

- **L'algorithme évolutif**

  https://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/poly/cours009.html

  Cet algorithme fonctionne de la même manière que la sélection naturelle.

  Fonctionnement :

  1. **Population initiale :** On crée un ensemble de solutions aléatoires au problème (la population initiale).
  2. **Évaluation :** Chaque solution est évaluée en fonction d'une fonction d'évaluation qui mesure sa qualité.
  3. **Sélection :** Les meilleures solutions sont sélectionnées pour se reproduire. Si une solution ne passe pas le critère (ici survivre) elle est supprimée.
  4. **Croisement :** Les solutions sélectionnées sont combinées pour créer de nouvelles solutions (comme la reproduction).
  5. **Mutation :** De petites modifications aléatoires sont apportées aux nouvelles solutions pour introduire de la diversité.
  6. **Répétition :** On répète plusieurs fois les étapes 2 à 5 jusqu'à ce qu'une solution satisfaisante soit trouvée ou qu'un critère d'arrêt soit atteint.

  Avantages :

  - Génération d'une grande diversité de solutions qui peuvent découvrir des chemins non évidents

  - Fonctionne bien même sur des fonctions d'évaluation non continues ou sans dérivées, ce qui est utile dans les problèmes auxquels il n'est pas possible de calculer un gradient.

  Inconvénients :

  - Coût en temps de calcul élevé, car il y a souvent de grandes populations et plusieurs générations pour converger.

  - Convergence vers une solution "suffisamment bonne" sans pour autant être optimaux dans certains cas.

  Exemple de jeu : **WorldBox**

- **La descente de gradient**
  https://www.ibm.com/fr-fr/topics/gradient-descent
  Fonctionnement :
  1. **Point de départ :** Choix d'un point de départ en utilisant des heuristiques (ou non)
  2. **Calcul du gradient :** On calcule le gradient en fonction de ce point.
  3. **Mise à jour :** Déplacement dans la direction opposée du gradient pour se rapprocher de la valeur minimale qui est le résultat optimal.
  4. **Répétition :** On répète plusieurs fois les étapes 2 et 3 jusqu'à ce qu'une solution satisfaisante soit trouvée ou qu'un critère d'arrêt soit atteint.

  Il faudra que l'on choisisse quelle descente de gradient choisir entre descente de gradient par lots, descente de gradient stochastique & descente de gradient par mini-lots.

  Avantages :
  - Peut converger plus rapidement vers une solution optimale, surtout lorsque la fonction est différentiable.
  - Différentes variantes de la descente de gradient permettent l'adaptation aux ressources disponibles.

  Inconvénients :
  - Le choix du point de départ peut influencer le résultat final, en particulier dans des espaces de solutions complexes où plusieurs minima locaux sont présents.
  - Un taux d'apprentissage trop grand peut empêcher la convergence en provoquant des oscillations, tandis qu'un taux trop faible rendra la convergence très lente.

  Je trouve que l'algorithme évolutif est plus complexe à mettre en place que la descente de gradient, mais plus efficace dans notre problème.

4. Étude des algorithmes choix de chemins

Ressource intéressante pour comprendre les algorithmes de "pathfinding" :
https://www.redblobgames.com/pathfinding/tower-defense/

- **La planification**

  Fonctionnement :

  On possède un état de départ. En ayant une séquence d'actions possibles, on essaye d'aller à un état final tout en gérant certaines contraintes.
  Pour cela, nous utiliserons différents algorithmes comme **Dijkstra**, **Bellman-Ford** ou **A***.

Cet algorithme pourra être couplé aux méthodes régissant le comportement des ennemis, pour avoir un chemin optimisé et des déplacements dynamiques. On le couplera dans un premier temps à des points de passages, par problème de simplicité, puis on pourra le coupler à une solution plus complexe, telle que les steering behaviours, qui permettra de suivre une trajectoire définie, tout en s'adaptant en temps réel aux interactions des ennemis avec les autres entités.

4.2. Explication des trois algorithmes

Algorithme de Bellman-Ford

Fonctionnement :

- Bellman-Ford est un algorithme qui peut trouver le chemin le plus court d'un sommet de départ à tous les autres sommets dans un graphe qui peut contenir des arêtes avec des poids négatifs.
- Il utilise une méthode de relaxation répétée : pour chaque arête, il vérifie si un chemin plus court vers un sommet est possible en passant par cette arête, et met à jour le coût minimum pour atteindre ce sommet si c'est le cas.
- Bellman-Ford répète cette étape pour chaque sommet du graphe, un nombre de fois égal au nombre de sommets moins un (|V| - 1) pour être sûr d'avoir trouvé le chemin le plus court dans tous les cas.

Complexité :

- La complexité en temps de Bellman-Ford est $O(V \times E)$, où V est le nombre de sommets et E le nombre d'arêtes.

Avantages :

- Contrairement à Dijkstra, Bellman-Ford fonctionne même dans les graphes avec des arêtes de poids négatif.
- Bellman-Ford peut détecter les cycles de poids négatif en exécutant une relaxation supplémentaire. Si un chemin plus court est encore trouvé, cela signifie qu'il y a un cycle de poids négatif.

Inconvénients :

- L'algorithme est plus lent que Dijkstra, particulièrement pour les grands graphes.

Algorithme de Dijkstra

Fonctionnement :

- Dijkstra est un algorithme de recherche du plus court chemin qui fonctionne uniquement sur des graphes avec des poids non négatifs.

- Il utilise une approche gourmande, en choisissant à chaque étape le sommet avec la distance minimale depuis le point de départ (non encore exploré) et en le marquant comme exploré.
- Ensuite, pour chaque voisin du sommet exploré, il met à jour la distance minimale de ce sommet s'il trouve un chemin plus court.
- L'algorithme s'arrête quand tous les sommets ont été visités, ou quand le sommet cible est atteint.

Complexité :

- La complexité de Dijkstra est $O((V + E) \times log(V))$.

Avantages :

- Pour les graphes sans arêtes négatives, Dijkstra est plus efficace que Bellman-Ford.
- Trouve le chemin le plus court pour des graphes pondérés sans poids négatif.

Inconvénients :

- L'algorithme ne fonctionne pas correctement si des arêtes de poids négatif sont présentes, car il suppose que les chemins deviennent de plus en plus longs, jamais plus courts.

Algorithme A*

Fonctionnement :

- A* est un algorithme de recherche du plus court chemin qui utilise une **fonction heuristique** pour guider la recherche. Cette heuristique (souvent notée $h(x)$) estime la distance restante entre un sommet actuel et le but.
- L'algorithme calcule une valeur $f(x) = g(x) + h(x)$, où $g(x)$ est le coût réel pour atteindre le sommet x depuis le point de départ, et $h(x)$ est l'estimation du coût restant pour atteindre la cible.
- A* choisit à chaque étape le sommet avec la valeur $f(x)$ la plus faible, ce qui lui permet de combiner la précision de Dijkstra avec l'efficacité d'une recherche dirigée.
- L'algorithme se termine lorsqu'il atteint le sommet cible.

Complexité :

- La complexité de A* dépend de la qualité de la fonction heuristique et de la densité du graphe. Dans le pire des cas, A* peut être aussi lent que Dijkstra $O((V + E) \times log(V))$

Avantages :

- Avec une bonne heuristique, A* peut être très rapide, puisqu'il réduit le nombre de nœuds explorés.
- A* permet d'incorporer différentes heuristiques, ce qui rend l'algorithme adaptable à de nombreux types de problèmes.

Inconvénients :

- : L'efficacité de A* dépend beaucoup du choix de l'heuristique. Si l'heuristique n'est pas admissible (ne sous-estime pas la vraie distance), A* pourrait ne pas trouver le chemin le plus court.
- A* utilise plus de mémoire que Dijkstra, car il doit stocker la totalité de l'espace de recherche dans le pire des cas.

5. Réseaux de neurones

Les réseaux de neurones sont un cas particulier, parce qu'ils agissent en tant qu'algorithme permettant l'amélioration des ennemis, mais également en tant que comportement des ennemis parce qu'ils peuvent également adapter le chemin/comportement adopté par les ennemis en fonction de leurs essais précédents.

6. Conclusion

Pour ce qui est du moteur de jeu, nous pensions choisir d'en créer un nous-mêmes en utilisant JavaFX car c'est le bon compromis entre performance et simplicité. Utiliser Swing nous aurait limité au niveau des performances et utilisé Slick2D aurait été trop compliqué pour un projet qui n'a pas pour cœur la création d'un moteur de Jeu.

Pour ce qui est des différents algorithmes d'évolution, nous allons en implémenter plusieurs et nous comparerons les performances de chacun. Pour ce qui est du déplacement, nous commencerons par un type de déplacement simple (Point de passage) pour ensuite en implémenter des plus complexes et efficaces tel que A* ou Steering behaviours, voir essayer de les combiner par la suite. Une fois que les algorithmes d'évolution seront fonctionnels, nous implémenterons d'autres solutions et nous les comparerons.

III) Conception

Itération 1 :

- Choix, compréhension et implémentation du moteur de jeu.

- Définition des éléments que nous retrouverons dans le jeu (défenses, ennemis, labyrinthe).
- **Produit Minimum Viable :** Moteur de jeu opérationnel (une fenêtre s'ouvre)

Itération 2 :

- Implémentation du déplacement des ennemis (point de passage).
- Créations de types d'ennemis, des types d'attaques pour préparer l'implémentation d'un algorithme d'évolution.
- **Produit Minimum Viable :** Génération d'une unité statique

Itération 3 :

Fin de l'implémentation du déplacement.

- Ajout d'algorithme de choix de chemins, sûrement Dijkstra pour commencer, étant donné que nous devrions avoir une unité capable de se déplacer dans le labyrinthe en fonction des points de passage donnés.
- Implémentation d'un premier algorithme d'évolution des ennemis.
- **Produit Minimum Viable :** Génération d'une unité et déplacement de cette dernière.

Itération 4 :

- Finalisation de la première version avec un algorithme de déplacement, de choix de chemin et d'évolution fonctionnelle.
- Préparation de la soutenance
- **Produit Minimum Viable :** Déplacement et évolution fonctionnels

Itération 5 :

- Ajout d'un algorithme de déplacement plus sophistiqué
- **Produit Minimum Viable :** Déplacement avancé des ennemis

Itération 6 :

- Implémentation de l'algorithme d'évolution sur les défenses
- **Produit Minimum Viable :** Défenses adaptatives

Itération 7 :

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