Projet PJI 110 : Tower Defense

Etudiants et Auteurs : CHARNEUX Dimitri et LEPRETRE Rémy
Encadrant : ROUTIER Jean-Christophe

Sujet : Création d'un jeu de Tower Defense en 2 Dimensions en Java

Master 1 Semestre 2 INFORMATIQUE 2016

SOMMAIRE

Remerciements

Avant de commencer notre rapport, nous tenons à remercier les personnes qui ont contribué à la réussite de notre projet.

Tout d'abord, nous souhaitons remercier toute l'équipe pédagogique de l'Université de LILLE1 pour leurs enseignements pendant ces quatre années qui nous ont amenés jusqu'à la fin de ce Master.

Enfin, nous tenons à remercier tout particulièrement Monsieur ROUTIER Jean-Christophe pour nous avoir suivis tout au long de la réalisation de ce projet et pour l'aide qu'il nous a apportée.

Introduction

Lors de notre première année de Master INFORMATIQUE, nous avons eu l'opportunité de réaliser un projet en autonomie qui nous tenait à cœur.
Nous avons choisi de réaliser un jeu de type Tower Defense basé sur celui déjà existant Dungeon Defender II.

Un Tower Defense consiste à défendre un certain point, que nous appellerons ici le nexus, contre plusieurs vagues d'ennemis qui avancent pour tenter de le détruire.
Pour défendre ce nexus, le joueur a la possibilité d'incarner un personnage et de poser des défenses afin d'empêcher les vagues d'ennemis d'atteindre le nexus.

Nous avons décidé de réaliser ce projet en JAVA en utilisant la librairie Swing. Mais contrairement au jeu d'origine, le nôtre sera réalisé en deux dimensions.

Dans un premier temps, nous allons présenter plus en détail notre projet. Ensuite, nous détaillerons les principaux points du développement de notre jeu. Puis, nous expliquerons les différentes possibilités d'évolution et d'amélioration de ce Tower Defense. Enfin, nous terminerons par une conclusion de cette expérience.

I/ Présentation du projet

Nous avons décidé pour ce PJI de créer un jeu de type Tower Defense. Pour vous expliquer ce projet, nous allons tout d'abord vous détailler ce qu'est un Tower Defense avant de nous concentrer sur les spécificités de notre jeu.

Un Tower Defense est un jeu dont le but est de défendre un point, le *nexus*, face à plusieurs vagues d'ennemis. Pour ce faire, le joueur a la possibilité de poser des défenses pour bloquer l'avancée adverse. Les défenses du joueur peuvent être de plusieurs types, des barrières pour simplement ralentir la progression ennemie ou des tours qui pourront les attaquer afin de les détruire. Le joueur peut utiliser son personnage pour attaquer directement les adversaires.

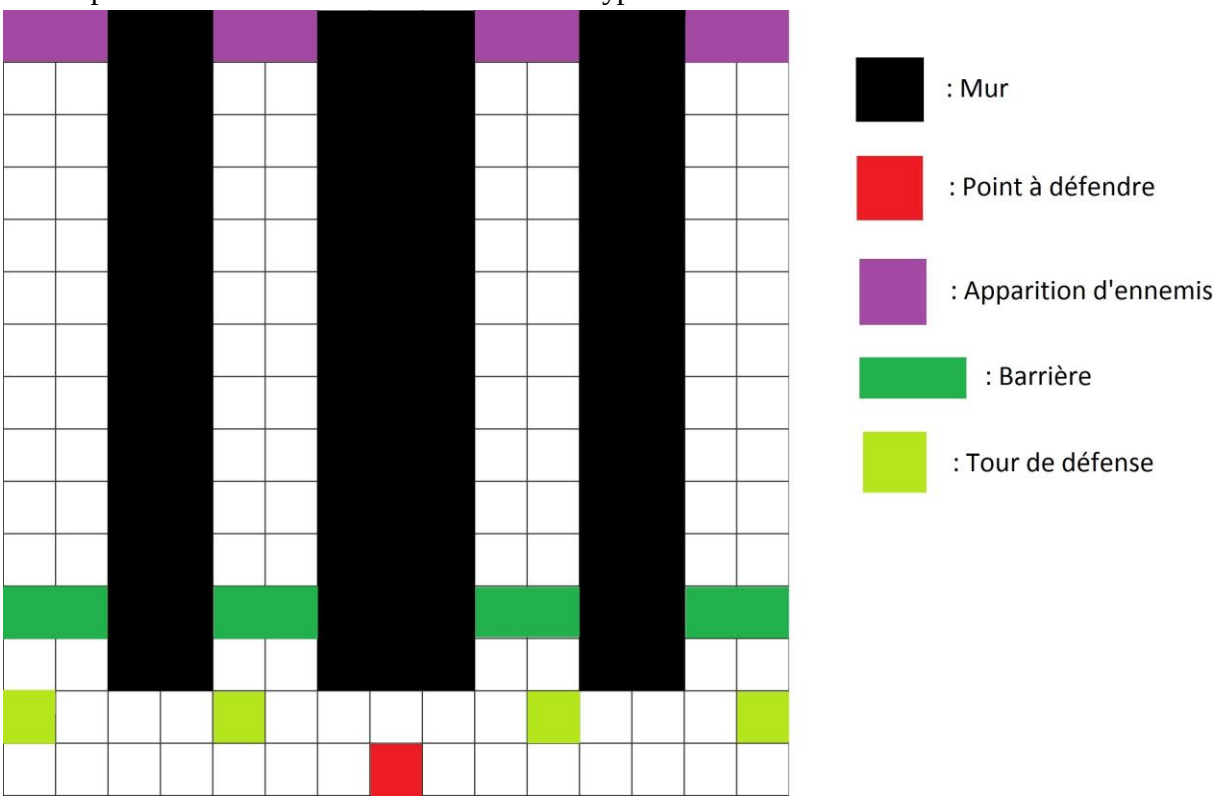
Une partie s'arrête quand le nexus a été détruit ou quand le joueur a réussi à détruire tous les ennemis en gardant le nexus en vie.

Le jeu se déroule en plusieurs ***tours***. Un tour est une unité de temps durant laquelle chaque entité pourra exécuter une action. Nous avons choisi d'appliquer les dégâts causés par une entité à la fin de chaque tour pour éviter les déséquilibres et permettre à toutes les entités d'attaquer.

Le jeu se déroule sur plusieurs cartes qui ont les mêmes caractéristiques :

- le ***nexus*** que le joueur doit à tout prix défendre
- des ***murs*** pour délimiter la zone
- les ***chemins*** que les vagues ennemies vont emprunter.

Vous pouvez voir ci-dessus le schéma type d'une carte de notre Tower Defense.



Dans ce texte, nous appellerons « entités » l'ensemble comprenant les ennemis, le personnage du joueur ainsi que les défenses.

Le joueur peut manipuler un personnage pour attaquer les entités ennemies. Il peut déplacer son personnage avec les touches Z Q S D ou les flèches directionnelles en fonction de ses préférences. Il peut aussi attaquer en appuyant sur la touche espace.

Un personnage incarné par le joueur peut passer à travers ses propres défenses mais les entités ennemies ne le peuvent pas.

Ensuite, pour poser ses défenses afin de protéger le nexus, le joueur doit positionner le curseur de la souris sur la case où il souhaite placer sa défense, puis effectuer un clic gauche pour la poser.

Pour apporter plus de stratégie au jeu, nous avons instauré un système de pièces. A chaque entité ennemie tuée, le joueur gagne des pièces, celles-ci pourront être utilisées pour poser les défenses. Le joueur devra donc faire attention à son nombre de pièces pour pouvoir poser des défenses au bon moment.

II/ Travail réalisé

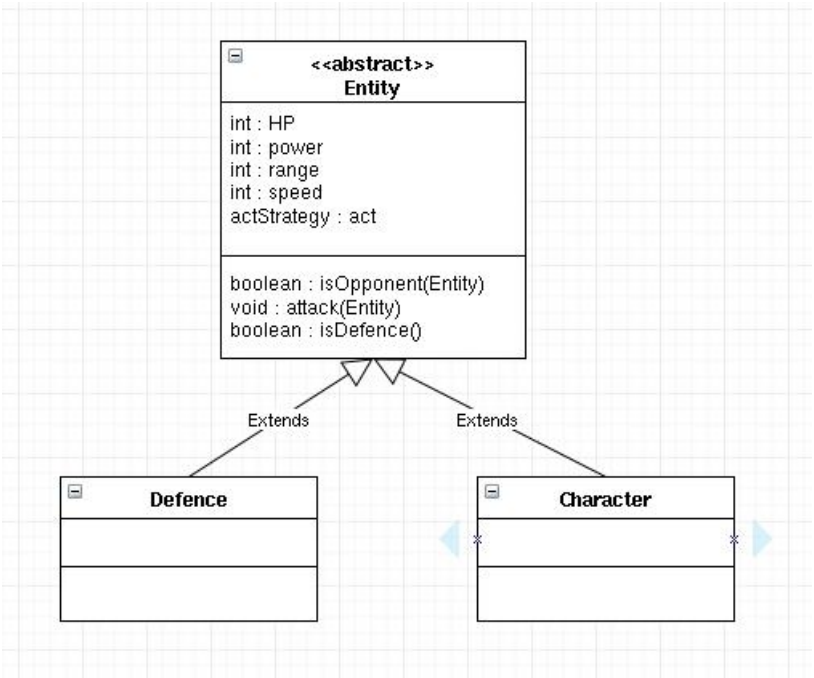
Nous avons commencé notre travail par l'établissement d'un diagramme de classes pour notre projet puis nous sommes passés à la partie développement.

a. UML

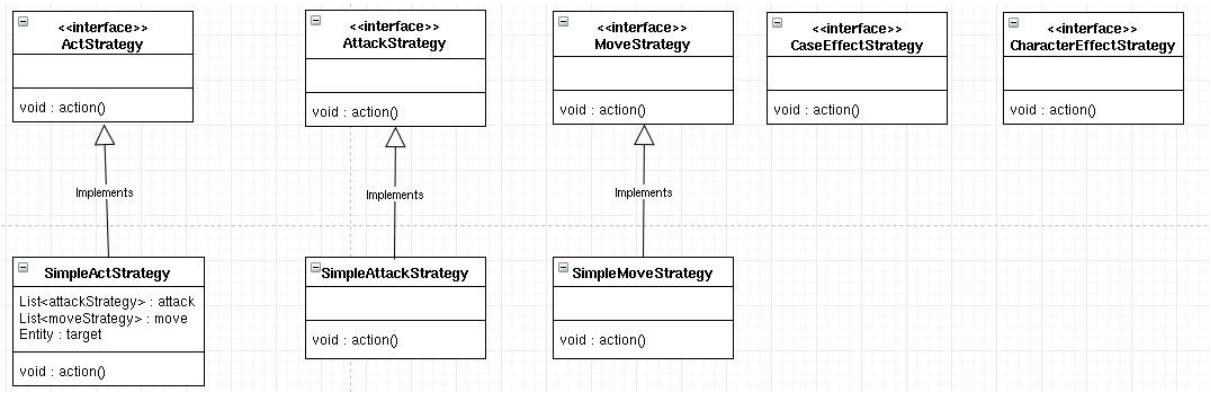
Avant de commencer à développer, nous avons donc établi le diagramme suivant afin de nous mettre d'accord sur les différentes classes à développer et de prévoir au mieux leur contenu.

Comme il est possible de le voir ci-dessous, les premières classes concernant les entités avec les `Characters` et les `Defences`.

Nous avons décidé de leurs donner une base commune qui est mise dans la classe abstraite `Entity`, puis de les différencier en deux classes, car on ne peut avoir qu'une `Defence` sur une case tandis qu'on peut avoir plusieurs `Characters`.

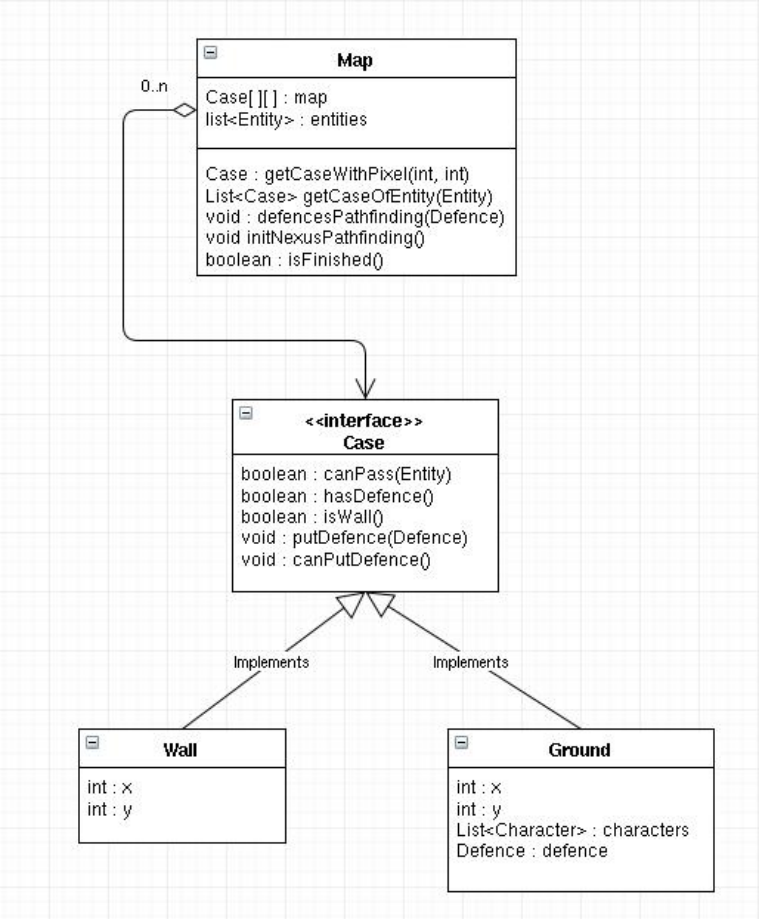


Chaque entité possède une stratégie qui est ici nommée *ActStrategy*. Toutes les *ActStrategy* sont chacune formée d'une, plusieurs ou aucune *AttackStrategys* et *MoveStrategys* (expliquées plus loin dans la partie Développement).



Il reste maintenant à parler de la carte (ici appelée *Map*) qui est formée d'une liste de *Cases* qui sont soit un *Wall* soit un *Ground*.

Les *Wall* sont des cases avec lesquelles aucune entité ne peut interagir et les *Ground* des cases sur lesquelles les entités se déplacent et sur lesquelles il est possible de poser une *Defence*.



Nous expliquons plus en détails dans la section suivante comment nous avons développé les différentes classes de ces diagrammes.

b. Développement

Dans cette partie, nous allons aborder les points principaux du développement de l'application dans l'ordre où ils ont été réalisés.

1. Entités

Nous avons commencé par le développement des différentes entités qui peuvent intervenir dans le jeu.

Il y a plusieurs types d'entité qui ont tous en commun les caractéristiques suivantes :

- les ***points de vies*** qui indiquent l'état de santé ;
- une ***puissance*** qui indique les dégâts occasionnés à chaque coup ;
- une ***portée*** qui indique à quelle distance une entité peut toucher sa cible.

Chaque type d'entité a un comportement propre. Il peut vouloir avancer vers le ***nexus*** sans se préoccuper des autres défenses, il peut attaquer chaque défense à sa portée ou cibler ses attaques sur un certain type de défense.

Les entités attaquent depuis le centre de leur image, mais peuvent être touchées sur la totalité de celle-ci. Elles ne peuvent se déplacer qu'horizontalement ou verticalement.

Toutes les entités, hormis les défenses, possèdent un attribut ***vitesse*** qui détermine le nombre de cases que va parcourir cette entité en un tour.

Ces entités se divisent en deux types :

- `Character`, qui regroupe d'un côté les différents personnages que le joueur va pouvoir incarner et de l'autre les monstres que celui-ci devra affronter ;
- `Defence`, qui représente les défenses qui seront posées par le joueur dans le jeu. Le nexus est une défense ne pouvant pas attaquer ni être placée par le joueur.

Plusieurs raisons nous ont incités à créer deux classes différentes pour les `Characters` et les `Defences`. Tout d'abord, une case peut contenir une liste de `Characters`, mais une seule `Defence`. En optant pour une seule classe, nous aurions pu mettre une `Defence` dans la liste de `Characters` ou un `Character` comme une `Defence` ce qui n'est pas souhaitable dans notre jeu. Ensuite, `Defence` et `Character` ont une différence au niveau du déplacement, une `Defence` est immobile tandis qu'un `Character` peut se déplacer, l'utilisation des deux classes permet également de prendre en compte ce point.

Maintenant que nous savons de quoi sont constituées les entités, nous allons voir comment leur comportement a été géré.

2. Stratégies

Nous nous sommes attaqués au développement l'intelligence artificielle des entités, qui est un point qui nous a pris beaucoup de temps. Pour ce faire, nous avons décomposé le comportement des entités en ***stratégies*** qui vont gérer les actions exécutées par l'entité. Il existe trois types de stratégie :

- les `moveStrategy` sont les stratégies utilisées par les entités afin de se déplacer (ex : avancer tout droit), le déplacement d'une entité est basé sur la ***vitesse*** couplée à une variable globale qui définit la distance en pixels parcourue en un tour;
- les `attackStrategy` sont les stratégies que vont employer les entités ennemies pour attaquer (ex : attaquer l'ennemi le plus proche). La détection des monstres à attaquer dépend de la caractéristique ***portée*** de chaque `Character` qui est une distance en pixels. La `map` possède une liste de tous les personnages et cette liste est parcourue afin de déterminer la distance entre le monstre attaquant et le monstre de la liste en cours de parcours afin de savoir s'il est à portée d'attaque ou non ;
- enfin les `actStrategy`, sont les ***stratégies*** qui définissent le déroulement principal de l'IA d'un monstre (ex : aller directement jusqu'au ***nexus*** pour le détruire). Les `actStrategy` analysent l'environnement (ennemis, alliés, chemins) de l'entité qui leur est associée et choisissent si l'entité doit attaquer ou se déplacer. Elles appellent ensuite la méthode `action` de `l'attackStrategy` ou de la `moveStrategy` pour que l'entité effectue l'action souhaitée.

L'IA d'une entité est donc définie par plusieurs `attackStrategys` et `moveStrategys` qu'il faut donc préciser. Elle possède aussi une `actStrategy` pour définir la corrélation qui existe entre `attack` et `move`.

Par exemple, si nous voulons créer une entité qui attaque en priorité le ***nexus***, nous allons utiliser une `moveStrategy` qui va avancer dans la direction du ***nexus***. L'`actStrategy` va regarder si l'entité peut avancer et appeler la `moveStrategy` pour effectuer le déplacement. Si l'entité ne peut pas avancer, `l'actStrategy` va appeler `l'attackstrategy` pour attaquer la `defence` qui lui bloque le passage.

Si nous voulons à présent créer une entité qui va attaquer les `defences` qui sont proches d'elle, nous allons garder la même `attackStrategy` mais utiliser une `actStrategy` différente. Celle-ci va regarder s'il y a une `defence` proche et appeler une `moveStrategy` qui va la diriger vers cette `defence`. Si aucune `defence` n'est assez proche, elle va donc réutiliser la même `moveStrategy` que l'entité qui vise le ***nexus***.

L'utilisation des `actStrategy` nous permet donc de réutiliser plusieurs ***stratégies*** pour obtenir des comportements différents.

Le déplacement de l'entité gérée par le joueur a nécessité des ***stratégies*** un peu différentes. L'`actStrategy` utilisée va capter les frappes au clavier pour savoir ce que doit faire l'entité. Si la touche frappée est l'espace, l'`actStrategy` va appeler une `attackStrategy` qui va attaquer l'entité ennemie la plus proche du personnage. Nous pouvons envisager par la suite d'attaquer dans la direction où le personnage pointe.

Si le joueur appuie sur les touches Z, Q, S ou D (ou les flèches directionnelles), l'`actStrategy` va appeler la `moveStrategy` en lui indiquant la direction souhaitée par le joueur. La `moveStrategy` va ensuite déplacer le personnage dans la direction voulue en fonction de sa vitesse et des obstacles sur sa route.

Maintenant que nous avons vu le fonctionnement des ***stratégies***, nous allons aborder comment la carte du jeu a été implémentée.

3. Carte

La Carte représente le terrain de jeu sur lequel va s'effectuer la partie et sera composée d'un nombre de `cases` de même taille préalablement définie par le niveau en cours.

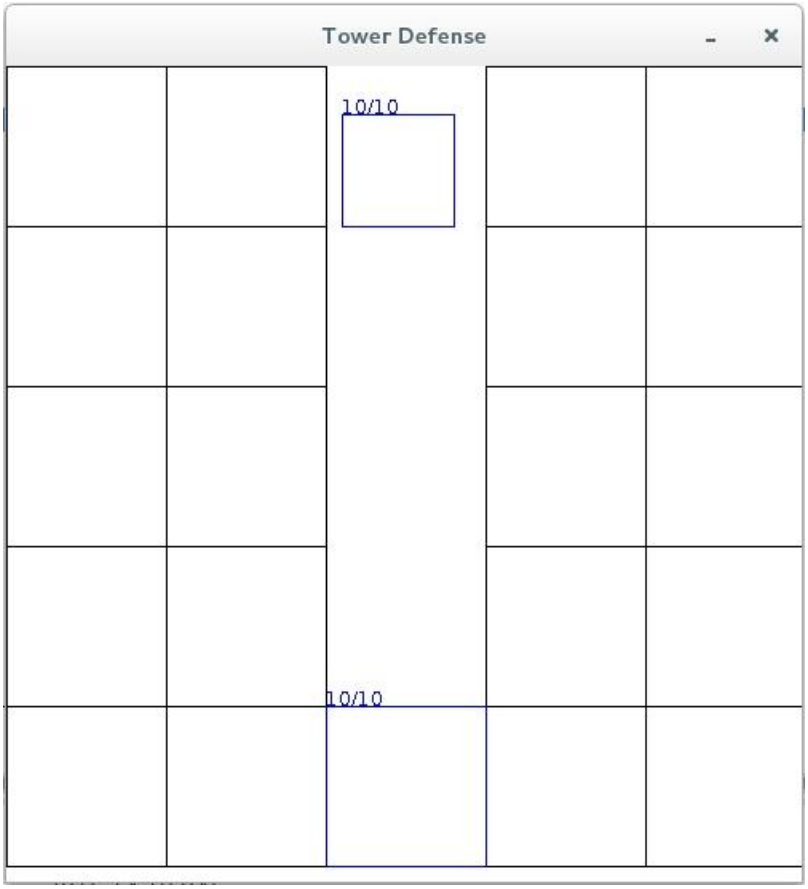
Pour dénommer la carte, nous utiliserons désormais le terme « `Map` ».

Sur la `Map`, les coordonnées horizontales sont les x et verticales les y. Le point en haut à gauche de la fenêtre est de coordonnée (0,0).

On distingue les cases `Ground` où les entités peuvent se déplacer et sur lesquelles le joueur peut placer des `defences`, une `defence` prend au minimum une `case` et on ne peut poser qu'une seule `defence` par `case`, et les cases `Wall` à travers lesquelles aucun personnage ne peut passer et sur lesquelles il est impossible de poser des `defences`.

La `Map` contient la liste de toutes les entités encore en vie et les `cases` contiennent quant à elles une liste d'entités qui sont sur celles-ci ainsi que les `Defences` posées dessus.

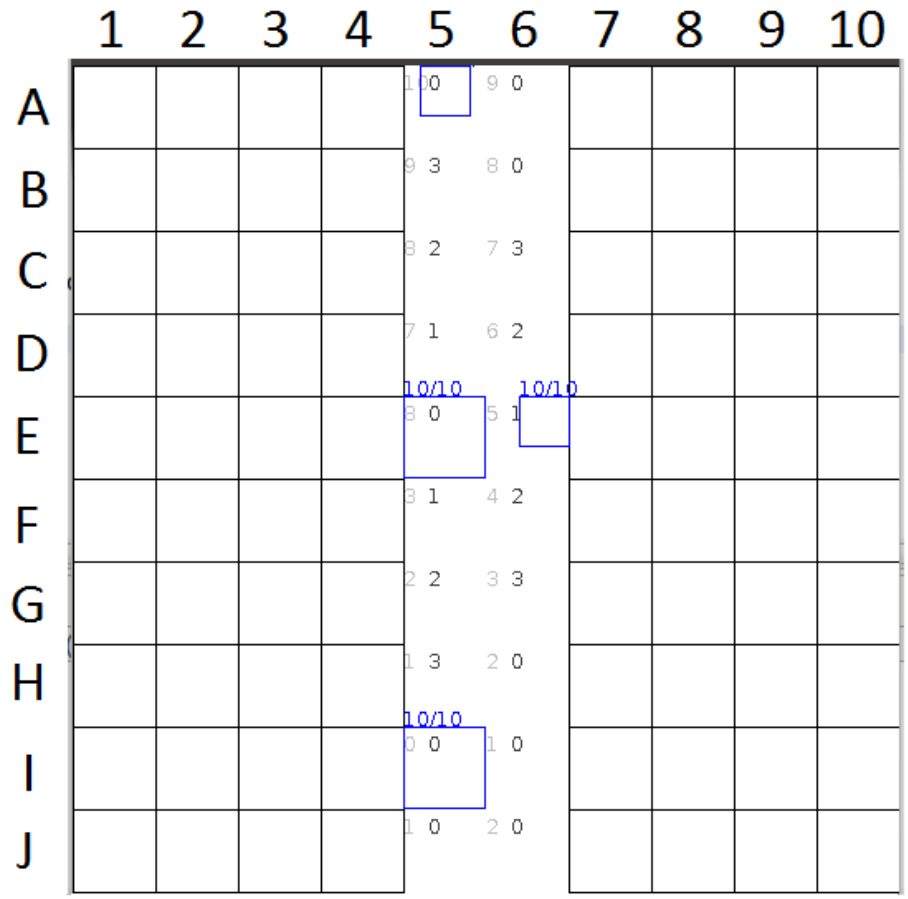
Nous avons réalisé une interface graphique pour le type `Map` où un `wall` est représenté par un carré au bord noir et une entité par un carré au bord bleu au-dessus duquel sont affichés les points de vie de l'entité. Un exemple de `Map` très simple est affiché ci-dessous.



Une fois la carte réalisée, nous avons géré le `pathfinding`, c'est à dire trouver une méthode pour permettre aux entités de calculer un chemin vers le ***nexus***. Nous avons choisi d'attribuer à chaque case de la carte un nombre qui indique à combien de cases elle se situe du ***nexus***. Pour attribuer ces nombres, nous avons utilisé un algorithme qui part du ***nexus*** et qui va attribuer 0 aux cases où il se trouve. L'algorithme va initialiser le numéro des autres cases à '-1' et ajouter la ou les cases où se trouve le ***nexus*** à une `LinkedList` (Une `LinkedList` est une liste où les données sont rangées selon le principe de premier entré, premier sorti). Ensuite, pour chaque case de cette liste, l'algorithme va chercher ses voisins et si leur numéro est égal à '-1' ou est plus grand que le numéro de la case actuelle +1, il mettra son numéro à celui de la case actuelle +1. Il ajoutera ensuite les voisins à la `LinkedList` et continuera jusqu'à ce que cette liste soit vide.

Cet algorithme sera lancé au début du niveau et à chaque fois qu'une `defence` est posée pour permettre d'actualiser les chemins.

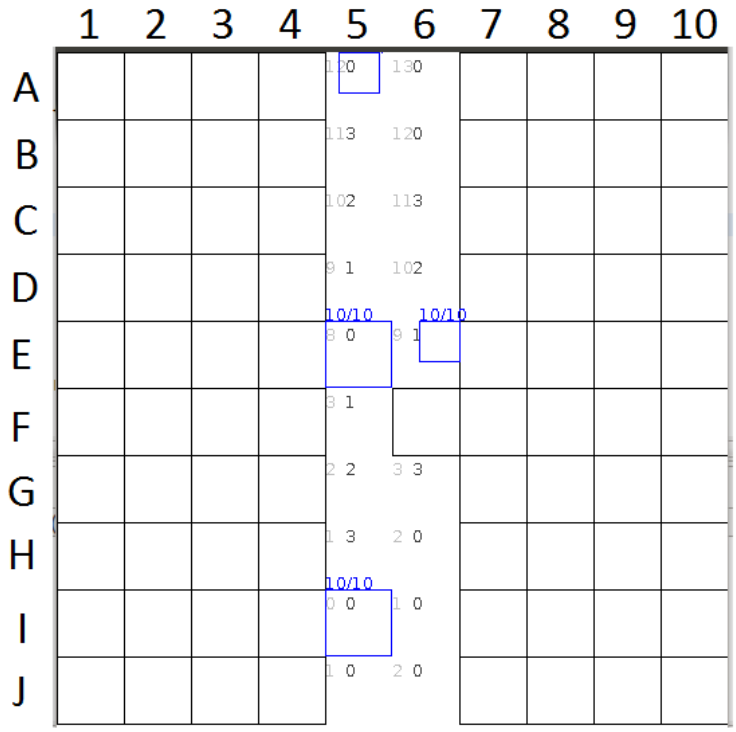
Pour vous montrer à quoi ressemble une carte après l'exécution de cet algorithme, voici une image qui nous montre la carte du jeu en début de partie. Le numéro créé est celui affiché en gris.



Nous nous sommes ensuite posés la question de savoir ce qu'il faut faire quand une `case` contient une `defence`. Si nous la considérons comme un `wall`, l'algorithme va l'ignorer et si la `defence` bloque un chemin, les `cases` se situant derrière ne seront pas numérotées et les entités ne sauront pas se déplacer. Si nous la considérons comme une `case` normale, les entités devront détruire la `defence` avant de passer ce qui leur fera perdre quelques tours alors qu'elles auraient peut-être pu passer par un autre chemin.

Nous avons donc choisi d'ajouter arbitrairement 5 points au numéro d'une `case` si elle contient une `defence`, ainsi, une entité privilégiera un chemin un peu plus long mais sans `defence`.

Après l'ajout d'un `wall` sur la case F6, le chemin est bloqué par la `defence` en E5. Les nombres calculés à partir de la `case` E5 sont donc incrémentés de 5.

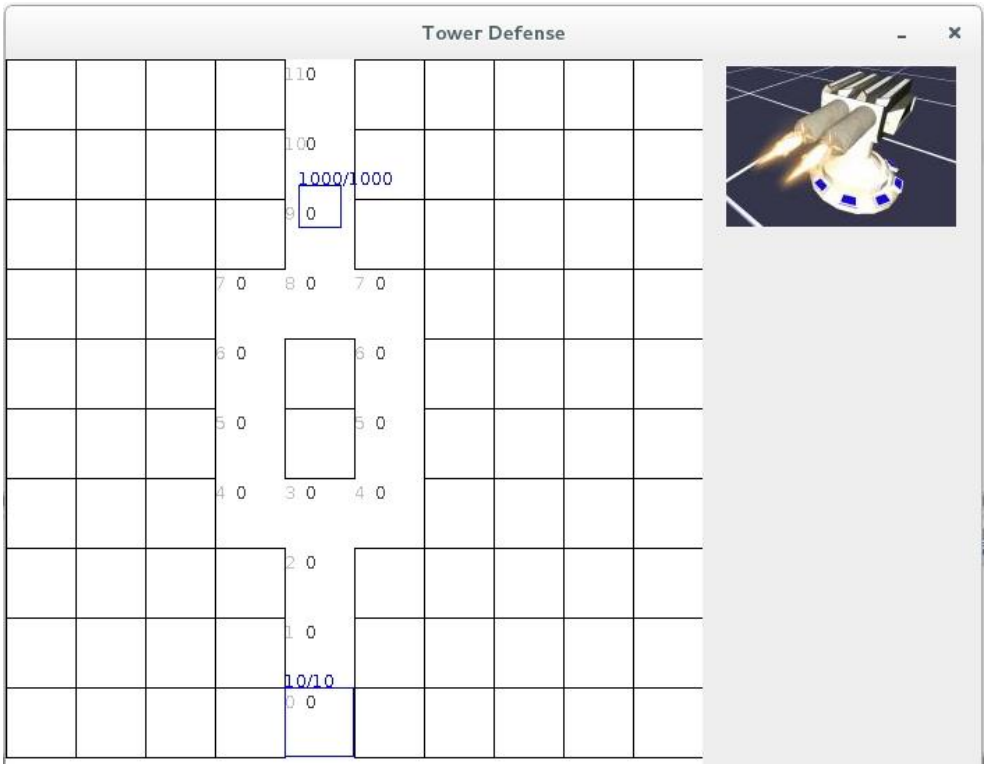


Ensuite, pour guider les entités préférant viser les *defences*, nous avons ajouté un second numéro qui indique le nombre de *cases* vers une *defence*. Ce numéro est représenté dans les deux images précédentes par le numéro noir en haut des cases. Il fonctionne de la même manière que celui du ***nexus*** sauf qu'il ne parcourt qu'un nombre prédéfini de *cases* (ici, ce nombre est 3), pour limiter les calculs ainsi que l'attraction des *defences*.

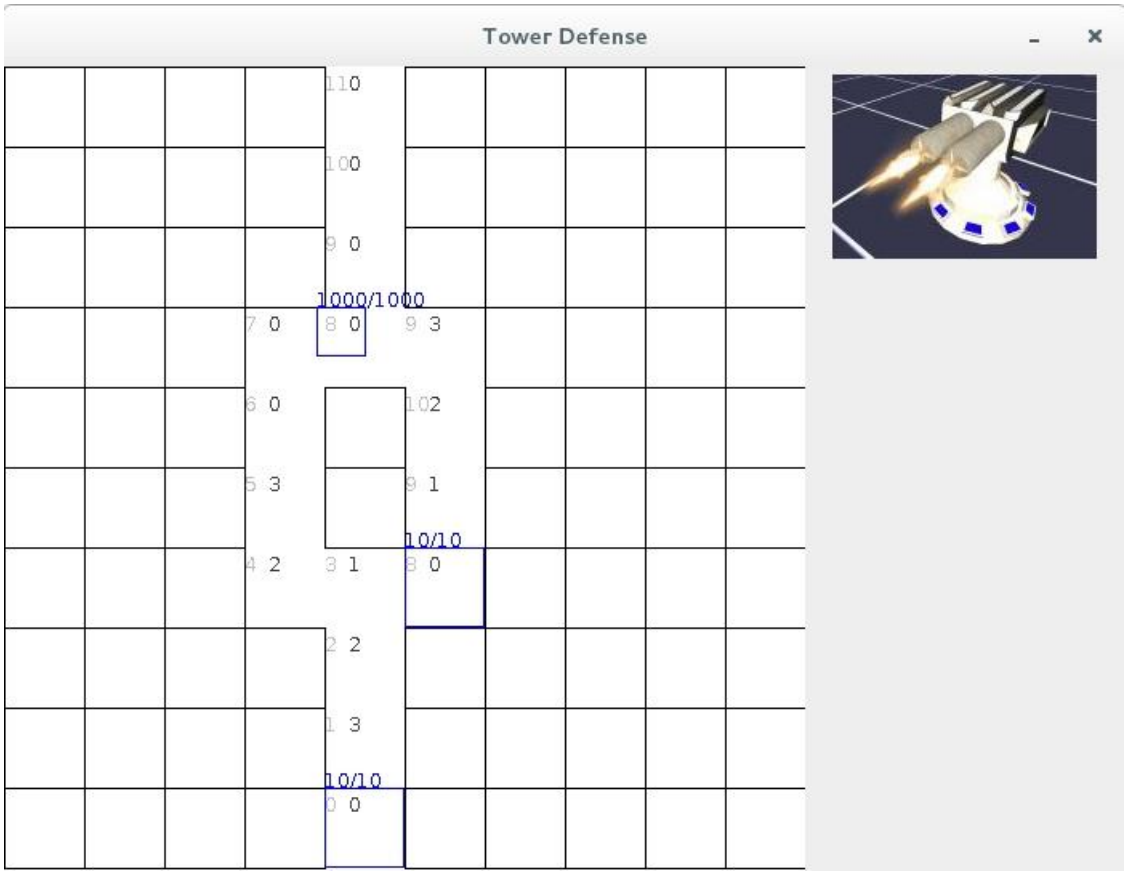
Pour la pose des *defences*, nous avons créé un bandeau positionné sur la droite de la *Map* qui contient l'ensemble des *defences* disponibles. Une infobulle apparaît au survol de chacune indiquant les caractéristiques de celle-ci ainsi que son coût en cristaux.

Afin de positionner une *defence* sur la carte, le joueur devra effectuer un clic gauche sur celle de son choix et, s'il possède suffisamment de cristaux, il pourra la placer avec un second clic gauche sur une *Ground* ne possédant pas déjà de *defence* et n'étant actuellement occuper par aucune entité.

Sur l'image située ci-dessous on peut donc voir à quoi ressemble l'interface pour ajouter une *defence*.



Tandis que sur l'image suivante on peut voir que lorsque l'utilisateur a fait un clic gauche sur une case qui n'est pas un *wall*, la *defence* a été ajouté et les *pathfinding* mis à jour.



Enfin, nous avons voulu simplifier la création de ces `maps`. Pour cela, nous avons utilisé des fichiers dans lequel une `map` est détaillée. Ces fichiers contiennent les `cases` présentes dans la `map`. Les `walls` est représenté par le symbole « X », une case `Ground` par le symbole « o » et le ***nexus*** par le symbole « N ». Au lancement d'un niveau, le programme va donc lire le fichier associé et créer la `map` avant de lancer la partie.

Maintenant que nous vous avons expliqué le développement de notre jeu, nous allons vous présenter le déroulement d'une partie.

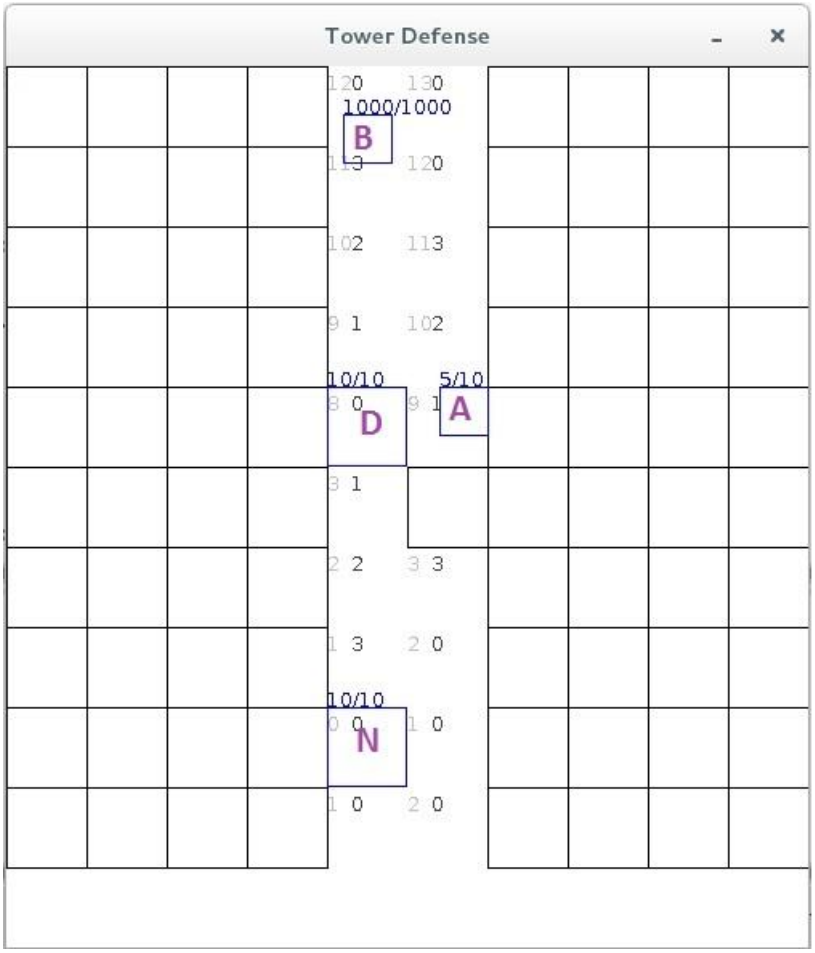
III/ Démonstration

Nous allons maintenant faire une petite démonstration du déroulement d'une partie sans intervention de la part du joueur.

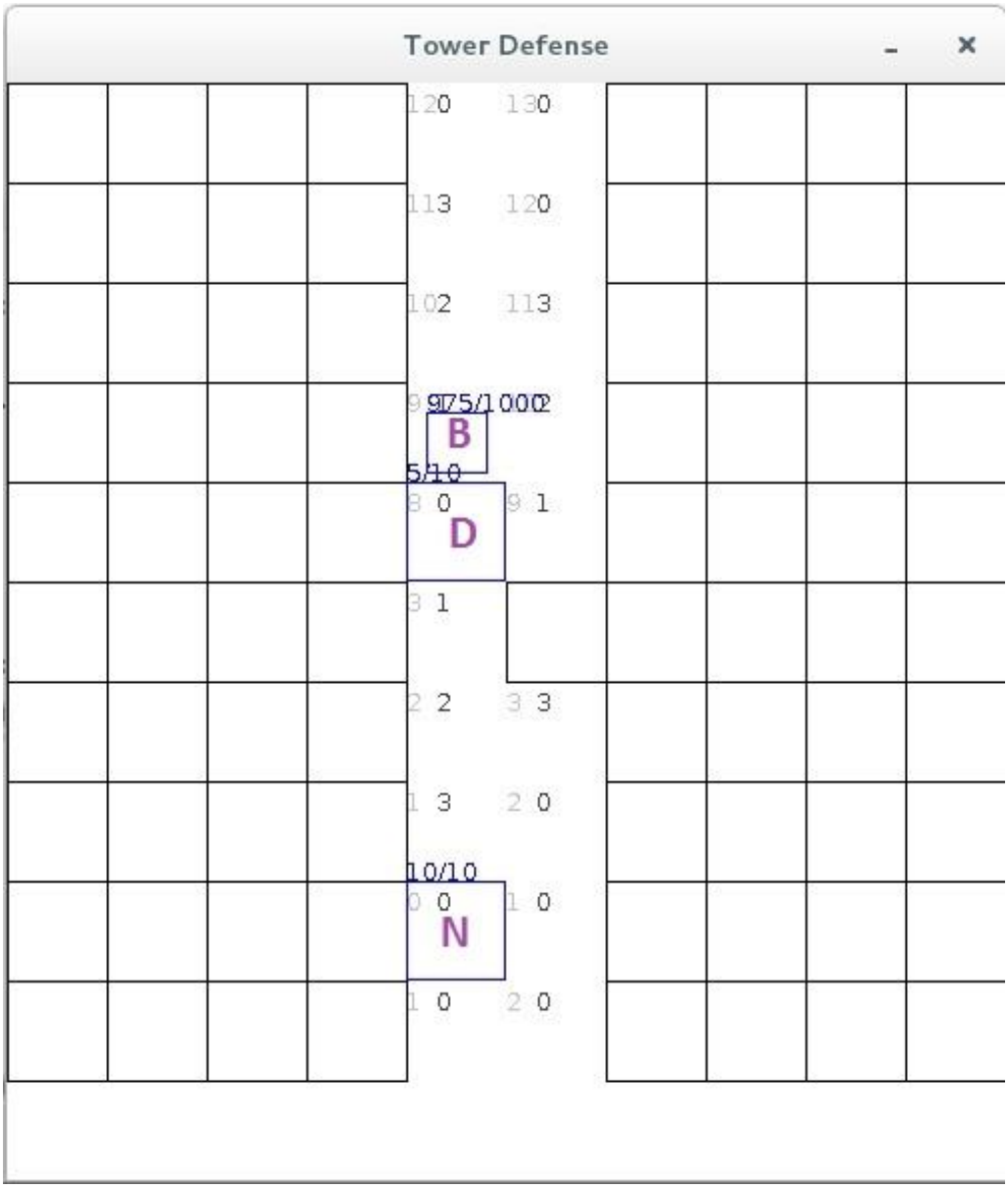
Tout d'abord, sur la `map`, il y a deux entités ennemies: une avec 10 points de vie et l'autre avec 1000 points de vie. L'entité ennemie la plus faible (que l'on nommera ici A) n'a aucune ***stratégie***, ni de déplacement, ni d'attaque (comme le ***nexus***). Quant à l'entité ennemie la plus forte (que l'on nommera ici B), elle a une ***stratégie*** de déplacement cherchant à aller attaquer le ***nexus*** le plus rapidement possible mais si elle se retrouve bloquer attaque l'entité ennemie la plus proche. La `defence` possède une ***stratégie*** d'attaque visant l'ennemi le plus proche.

Sur cette `map` il y a également une `defence` positionnée à côté de A et le ***nexus*** qui est situé tout en bas de la carte.

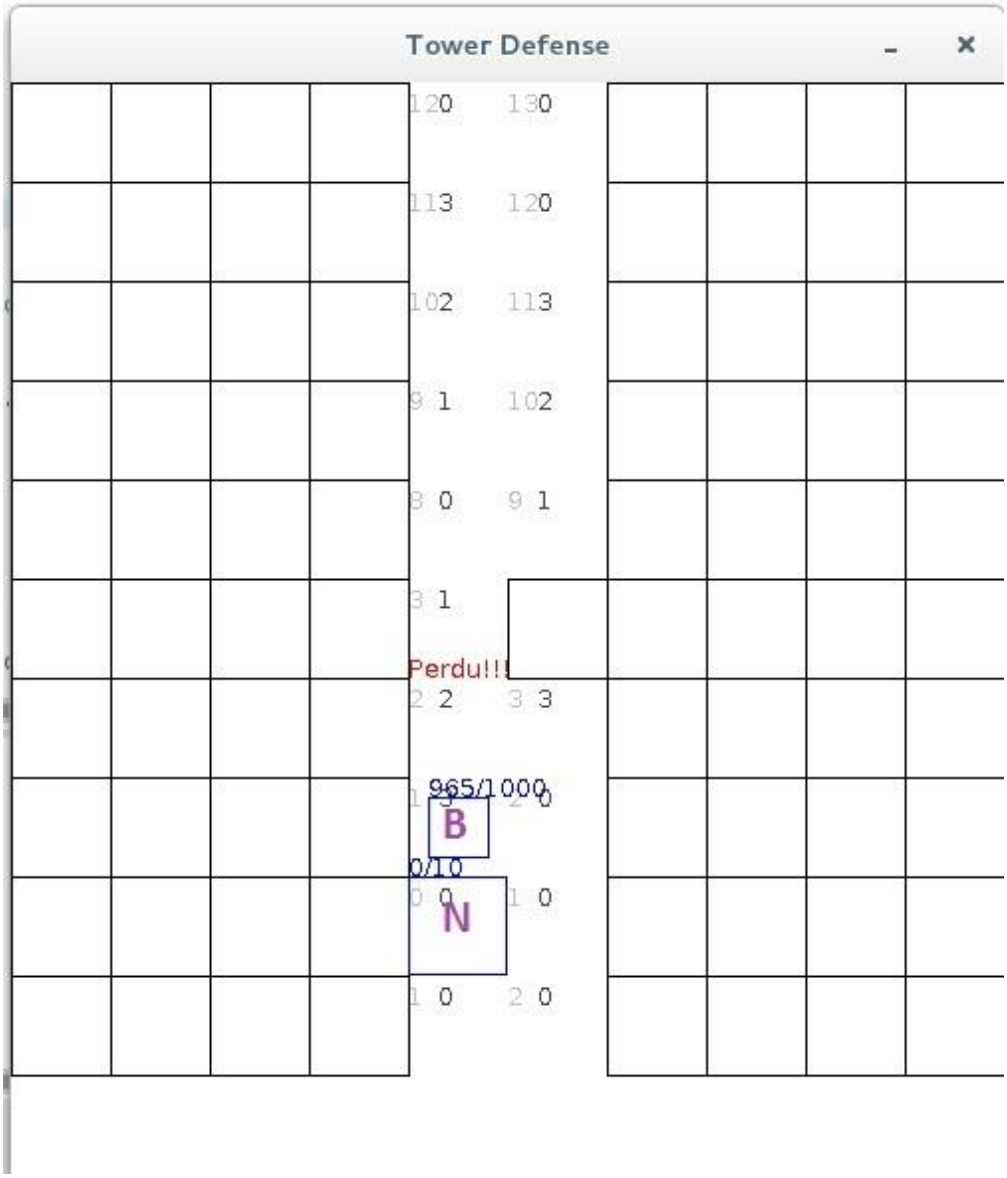
Comme on peut le voir sur l'image ci-dessous, A se fait attaquer par la `defence` pendant que B se dirige vers le bas.


Tower Defense
1000/1000
B
10/10
5/10
D
A
10/10
N

Maintenant, sur l'image ci-dessous, A a été détruit par la `defence` et B a avancé et cherche à contourner la `defence` pour aller directement attaquer le ***nexus*** mais la `defence` posée ainsi que le `wall` lui bloque le chemin. B se voit donc contraint d'attaquer la `defence` alors que celle-ci est déjà en train de l'attaquer.



B ayant fini par détruire la `defence` il peut maintenant se diriger vers le ***nexus*** pour le détruire également. Une fois arrivé devant, il commence à l'attaquer et lorsque les ***points de vie*** du ***nexus*** atteignent 0 la partie est terminée et le joueur est déclaré perdant car le ***nexus*** a été détruit.



Voilà pour le déroulement d'une partie simple, sans actions de la part du joueur. Nous allons maintenant vous présenter les perspectives d'évolution de notre jeu.

IV/ Perspectives d'évolution

Il est encore possible de faire évoluer notre jeu, nous allons maintenant voir les différents éléments qui pourraient être ajoutés ou modifiés à notre projet dans le futur.

a. Diversifier

Le premier élément à ajouter à notre jeu sera des nouvelles `maps`. Nous allons créer différentes `maps` réparties par niveaux. Chaque niveau abordera un thème et aura des graphismes associés à celui-ci.

Nous pensons également à intégrer un éditeur de niveaux qui va permettre aux joueurs de créer leurs propres `maps` et les partager avec d'autres joueurs.

Nous voulons créer également d'autres ennemis ainsi que d'autres `defences`. Pour ce faire, nous allons ajouter d'autres ***stratégies*** d'attaque, de déplacement et d'action. Notre implémentation est telle qu'elle facilite la création de multiples ***stratégies*** en réutilisant les ***stratégies*** déjà implémentées ou en en développant de nouvelles. Nous pourrons ainsi ajouter des entités ennemies attaquant en priorité les `defences` et d'autres ciblant en priorité le personnage du joueur. Nous pensons également à faire de nouvelles `defences` qui cibleront les ennemis ayant peu de points de vie pour éclaircir un peu le champ de bataille. D'autres `defences` et adversaires seront ajoutés au fur et à mesure que nous les imaginerons.

b. Sorts et équipements

Un autre élément à ajouter serait un système d'équipement. Les équipements portés par le joueur modifieront ses statistiques pour lui permettre de venir à bout de niveaux plus difficiles.

Ces équipements pourront être achetés dans une boutique en échange d'or récupéré en terminant un niveau. Ils pourront également être lâchés par des entités que le joueur aura vaincues.

Pour les implémenter, il faut créer une classe `equipment` qui possédera des statistiques comme pour les entités (vitesse, attaque, etc). Une entité possédera une liste `d'equipments` qui influeront sur ses statistiques. Pour appliquer les statistiques des `equipments` à ceux des entités, il faudra additionner la valeur des statistiques des `equipments` à celle de l'entité dans les méthodes `get`() de cette dernière.

Nous envisageons également d'ajouter d'autres attaques pour le personnage contrôlé par le joueur. Il pourra alors réaliser des attaques de zone ou des attaques lui permettant de récupérer de la vie. Chacune de ces attaques coûtera au joueur des points de ***mana*** qui se rechargeront au cours du temps.

Ces attaques seront créées grâce à des `attackStrategys`. L'`actStrategy` d'une entité choisira alors quelle attaque exécuter en fonction de son environnement.

Pour le joueur, les nouvelles attaques seront exécutées grâce aux touches numériques qui seront détectées par l'`actStrategy` de l'entité.

c. Effets

Les attaques lancées par les entités pourront avoir des effets. Il existera plusieurs types d'effets appliqués aux attaques :

- le poison et le feu infligent des dégâts à chaque tour à l'entité touchée ;
- le froid peut ralentir les entités durant un laps de temps et le gel peut même les immobiliser ;
- l'effet confusion peut quant à lui retourner une entité contre les siens pendant quelques secondes.

Ces effets peuvent aussi être appliqués à des `cases`, ainsi, une `case` enflammée brûlera les entités se déplaçant dessus. Les effets appliqués à des `cases` seront actifs pendant un nombre prédéfini de tours.

Enfin, certaines entités qu'on appelle ***entités de soutien*** peuvent soigner ou booster temporairement leurs alliés. Ainsi, une entité peut voir son attaque, sa portée ou sa vitesse améliorée grâce à un allié situé à proximité.

Pour réaliser les effets, nous allons créer des ***stratégies*** de la même manière que pour les attaques. Ces `effectStrategys` contiendront un compteur qui indiquera pendant combien de tours ils seront actifs. Il faudra ajouter aux `cases` une liste `d'effectStrategy` qui pourrait être appliquées à la `case`. A la fin de chaque tour, les `effectStategy` vont être appliquées sur leurs `cases` et leurs compteurs seront décrémentés. Une fois le compteur à zéro, les `effectStrategys` vont être supprimées de leur `case`.

Conclusion

Lors de ce projet, nous avons réalisé un jeu de type « Tower Defense ».

Après ces quatre mois de projet, il est possible de créer une `Map` à partir d'un fichier et sur laquelle des monstres peuvent se déplacer selon différentes ***stratégies***, les calculs de `pathfinding` permettant de trouver le chemin optimal.

Le joueur a la possibilité de poser des `defences` et de déplacer son personnage sur la `map`.

Nous avons pu mettre en pratique plusieurs de nos connaissances acquises au cours de notre parcours universitaire.
En effet, nous avons pu utiliser les connaissances vues en IHM pour créer les interfaces graphiques, ceux vues en GL pour créer les UML et gérer correctement notre projet, ainsi que les compétences en algorithmiques pour gérer le `pathfinding` des entités.

Ce projet nous a également appris à gérer un travail en équipe sur un sujet qui nous tient à cœur. Étant en total autonomie, il nous a confronté à différentes situations qu'il nous a fallu gérer : la gestion de notre temps sur notre projet car celui-ci étant entremêlé parmi les différents TP que nous avions dans les autres matières et ensuite une répartition équitable du travail sur le projet afin que nous ayons la même quantité de travail.

Bibliographie

Jeu de référence :
https://dungeondefenders.com/2/

Calcul de pathfinding :
http://www.redblobgames.com/pathfinding/tower-defense/

Annexe

Exemple de fichier représentant une map :

```
10
10
XXXXoXXXXX
XXXXoXXXXX
XXXXoXXXXX
XXXoooXXXX
XXXoXoXXXX
XXXoXoXXXX
XXXoooXXXX
XXXXoXXXXX
XXXXoXXXXX
XXXXNXXXXX
```