Dossier du Projet (19-20)

Groupe : CORMY Joris, MARQUES DE JESUS Rémi, DIAS Thomas

Ce fichier reprend la description des documents demandés pour le projet, ainsi que quelques renseignements que vous devez fournir.
Vous devez le compléter et le mettre à disposition sur git.

Tous les diagrammes devront être fournis avec un texte de présentation et des explications.

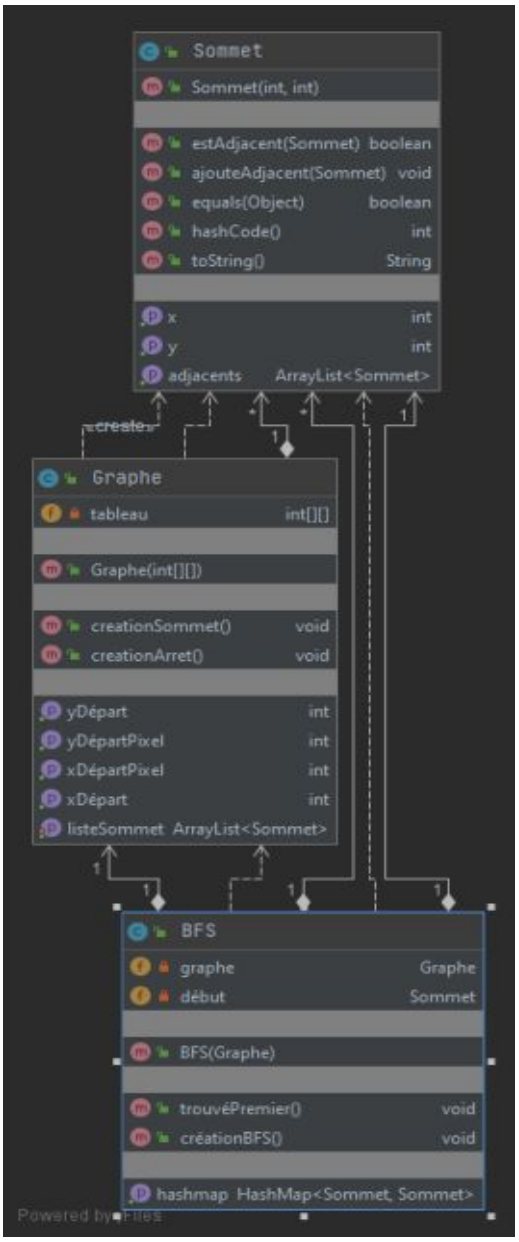
CPOO		IHM		Projet Tuteuré		Gestion de Projet	
diagrammes d'architecture	9	ergonomie et contenus	15	soutenance	10	document utilisateur	8
diagrammes de classes	9	programmation évènementielle et JavaFX	35	niveau et qualité du produit fini	15	Trello	6
diagrammes de séquence	9			notes de sprints	15	Git	6
POO	24						
Structures de données	6						
gestion des erreurs	5,5						
tests	5,5						
algos	17						
ampleur et qualités du code	25						
	110		50		40		20

Documents pour CPOO (sur 27)

1.1 Architecture (9 points)

Comme architecture pour notre jeu, nous avons séparé les différents éléments de celui-ci en différents packages, nous avons créé quatre classes abstract nommées “Tir”, “Tourelles”, “Ennemis” et “Pieges”, chacune d'elles étant étendues par des classes filles qui vont permettre de créer des variations dans les différents types d'entités. Ces classes mères contiennent la majeure partie des attributs nécessaires à chacun des objets créés à partir de ces classes, mais aussi des méthodes essentiels. Ces méthodes sont parfois sous forme abstraite car les différentes variations auront des comportements différents, ou alors les méthodes sont directement développées afin de réduire les répétitions de code.

1.2 Détails : diagrammes de classe (9 points)



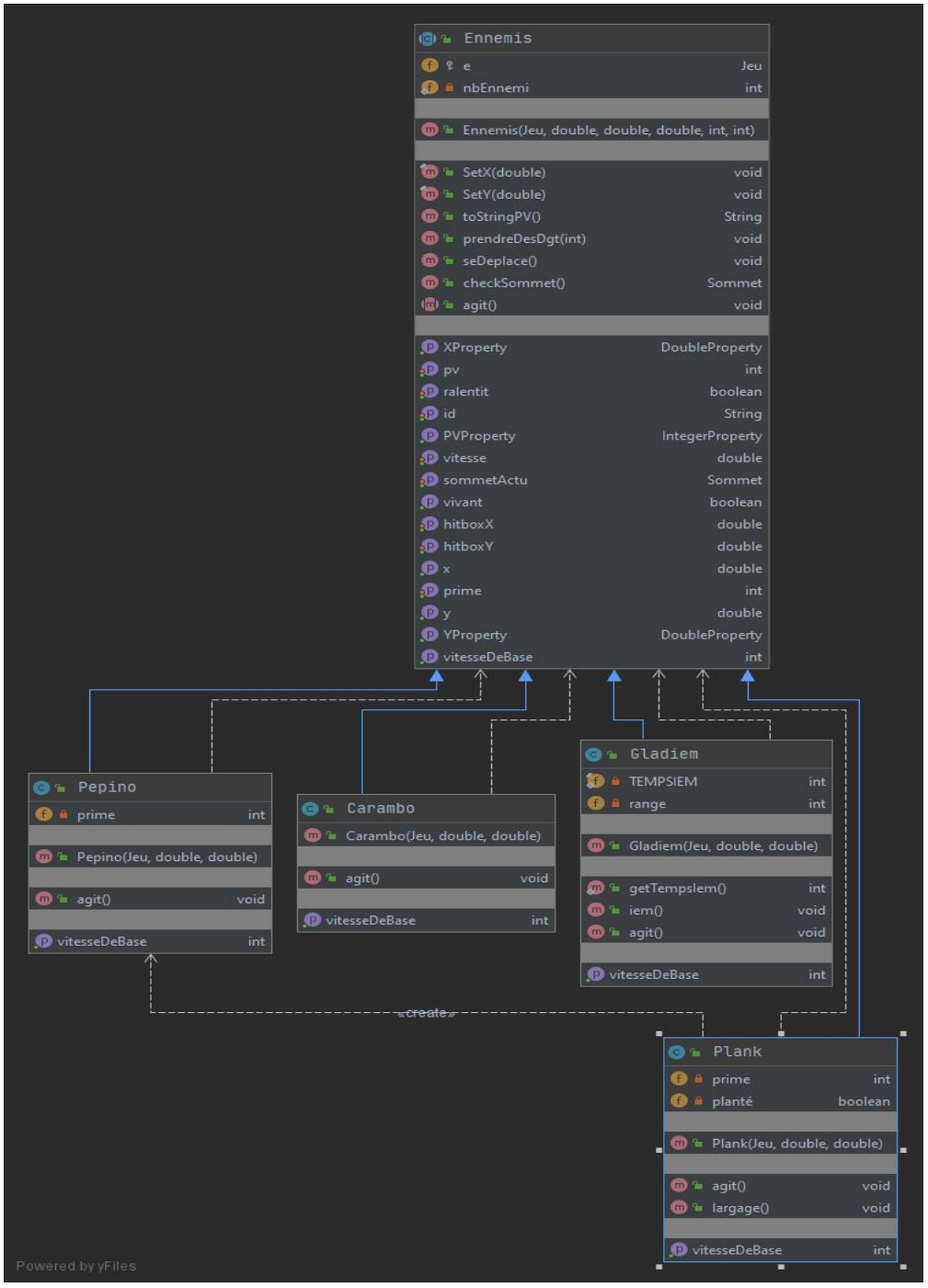
1) Le premier diagramme de classe comporte 3 classes importantes du programme : “Sommet”, “Graphe” et “BFS”, chacune liée aux autres.

   La classe “BFS” possède en attribut le graphe et le sommet de départ de l’algorithme ainsi que la hashmap qui permettra la synchronisation des sommets avec un sommet parent. Les différentes méthodes sont trouvéPremier(), qui permet de chercher le premier sommet qui servira de point de départ à l’algorithme dans le graphe, et creationBFS(), qui remplit la Hashmap.

   La classe “Graphe” a en attribut un tableau d’entier et une liste de sommets qui contient tous les sommets où les ennemis peuvent passer. La méthode creationSommet() va permettre de remplir la liste de sommet du graphe avec des

sommets. La méthode creationArret() quant à elle va permettre de remplir la liste adjacents de chaque sommet selon les différents sommets qui l'entourent.

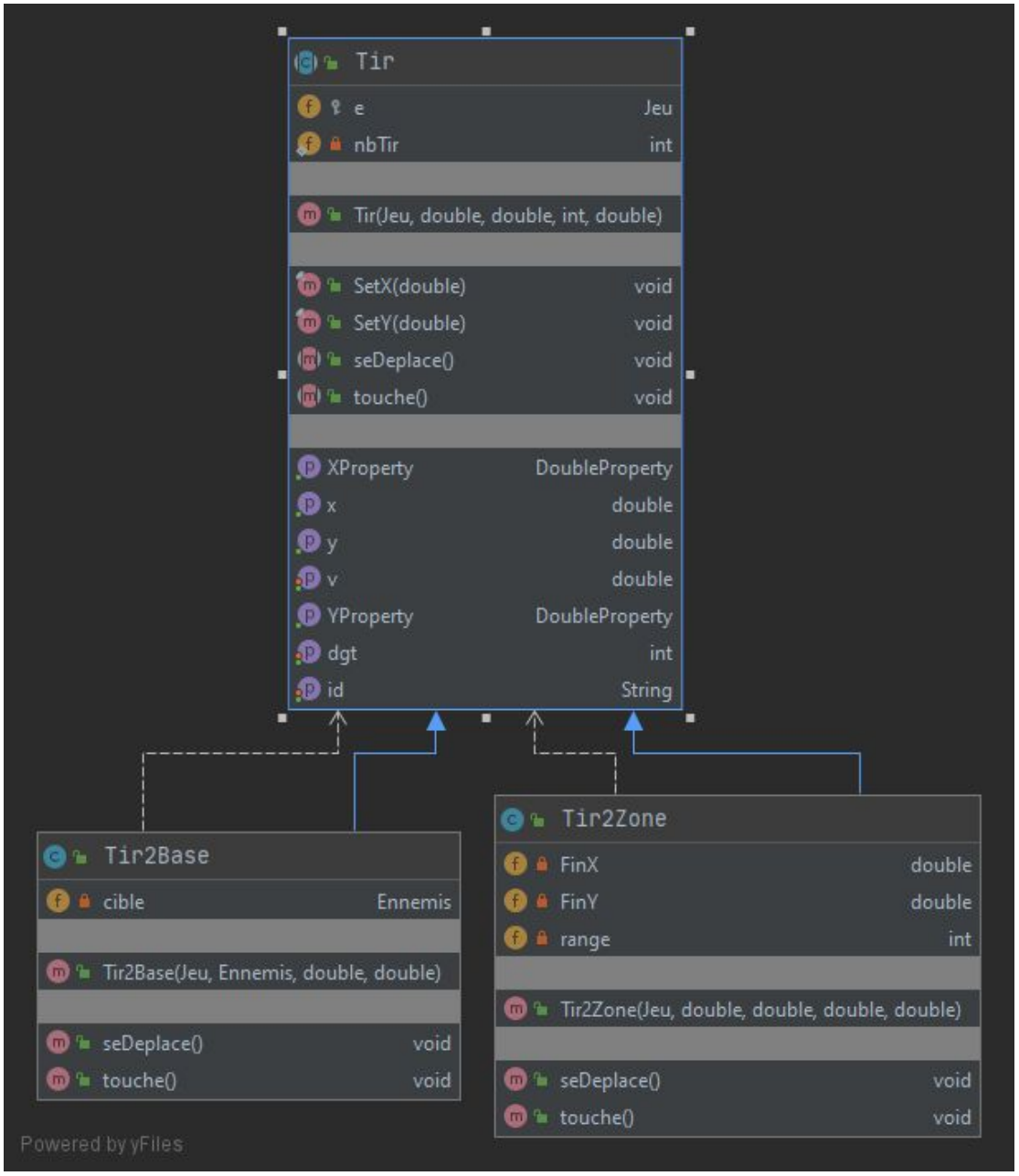
En effet, la classe "Sommet" a pour attribut un x et un y qui vont servir de coordonnées, et une liste d'adjacents afin que le sommet sache quels sommets l'entourent. La méthode estAdjacent() sert donc à définir si un sommet est à côté d'un autre, et ajoutAdjacent() est la méthode appelée pour ajouter un sommet adjacent à la liste.



2) On voit ici le diagramme de la classe abstraite “Ennemis” ainsi que les différentes classes d’ennemis qui en découlent. Ce type d’architecture se retrouve plusieurs fois dans notre projet avec les classes “Tourelles” et “Pieges” par exemple.

   On voit ici clairement que la classe mère “Ennemis” contient les différentes attributs de base communs à tous les ennemis, et cela malgré des variations de valeurs évidentes. Le constructeur de cette classe est par conséquent appelé par toutes les variations d’ennemies dans leur constructeur respectif. On remarque aussi la présence des méthodes prendreDesDegats() et seDeplace() qui sont communes à tous les ennemis et sont donc directement développées dans cette classe. La méthode agit() est déclaré sous forme abstraite car elle est nécessaire lors de l’appel des ennemis à chaque tour, mais le comportement des différents ennemis est variable, ce qui nous force à développer des méthodes différentes pour chacun d’eux.

   On remarquera notamment la Plank et la Gladiem ont respectivement les méthodes largage() et iem(), qui leur sont spécifiques car utilisées pour activer leur effet propre. Ces deux classes ont aussi des attributs supplémentaires, qui là encore ont trait à leur capacité spéciale.



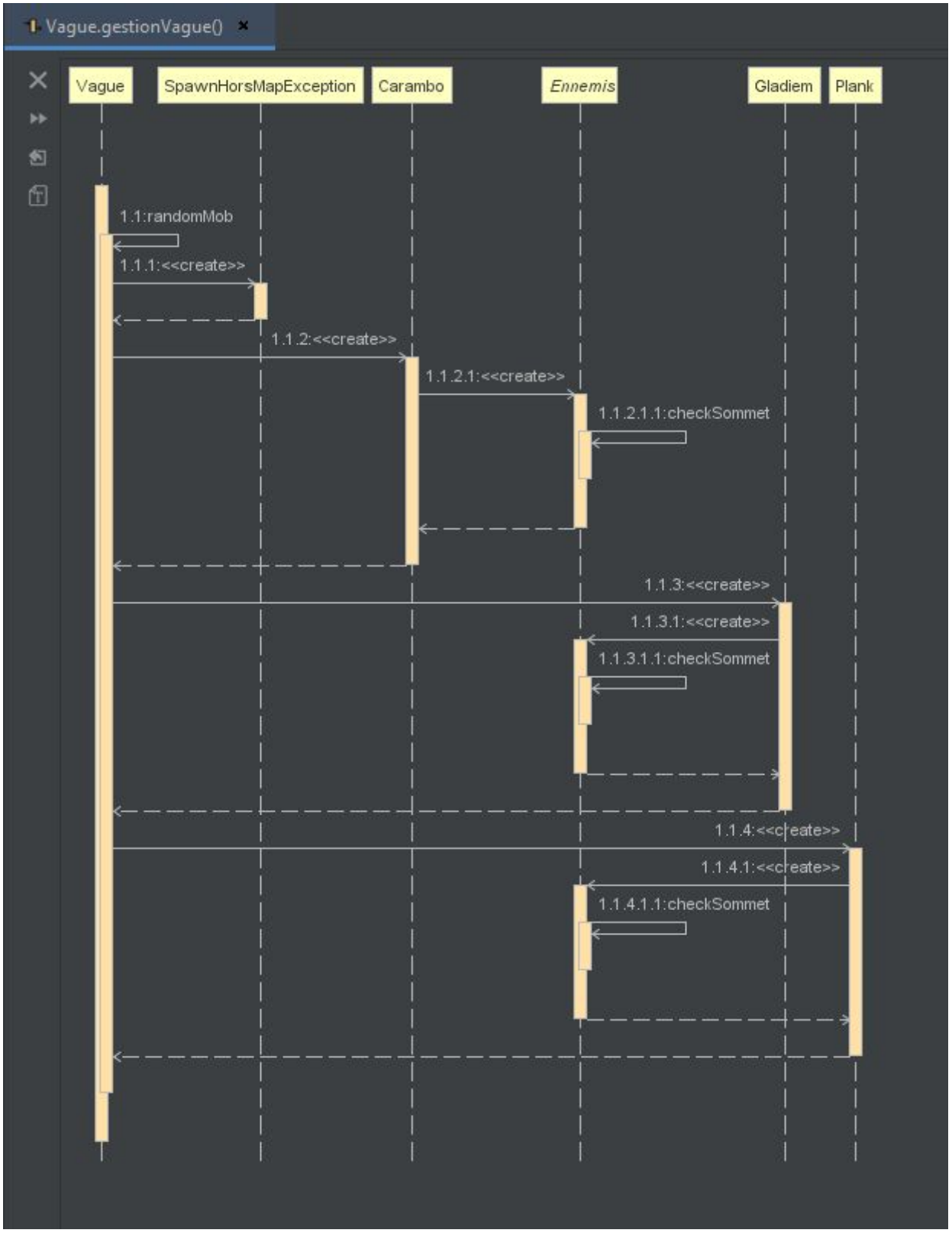
3) Voici le diagramme des classes relatives aux tirs. On retrouve un schéma similaire aux classes relatives aux ennemis avec une classe mère et des classes filles pour les variantes de tirs. On observe la présence de deux types de tirs différents malgré la présence de trois tourelles. Cela est dû à la présence de la tourelle de ralentissement, qui va agir dans la zone autour d'elle de façon constante sur tous les ennemis à portée, sans envoyer un quelconque projectile.

   On peut remarquer une différence majeur entre les deux types de tirs en la présence d'un attribut cible pour le Tir2Base. Sa présence permet au tir de suivre un ennemi et donc de donner une propriété de tête chercheuse au tir, ce que nous n'avons pas voulu reproduire pour le Tir2Zone. Le fonctionnement de ces derniers est plus proche de celui d'un mortier qui viendrait s'écraser après un certain délai sur un point précis, tout en faisant des dégâts dans la zone autour du point

d’impact. Ainsi, le Tir2Zone va simplement sauvegarder les coordonnées vers lesquelles il doit aller, sans se préoccuper de suivre un ennemi.

Diagrammes de séquence (9 points)

Vous choisirez une ou deux méthodes intéressantes du point de vue de la répartition des responsabilités entre différentes entités du programme. Vous utiliserez des diagrammes de séquence pour expliquer l’exécution de ces méthodes



Avec ce diagramme de séquence nous pouvons analyser le déroulé de l'exécution de la méthode gestionVague() présente dans la classe “Vague”.
Le déroulé est le suivant :

1) On appelle la méthode randomMob() pour créer un ennemi;
2) L’exception présente dans la méthode randomMob() va être testée;
3) Une fois l’exception testée et les possibles erreurs résolues, la création de l’ennemi peut continuer. On a donc un appel du constructeur de Carambo qui va lui même appeler le super() de la classe mère “Ennemis”;
4) Le super() de la classe “Ennemis” va appeler la méthode checkSommet() afin de vérifier que l’ennemis apparaîtra bien sur un sommet du BFS;
5) Une fois la méthode checkSommet() appelée, on retourne vers la méthode randomMob() qui va continuer la création des autres types d’ennemi de la même manière que pour le Carambo (appel du constructeur du type de l’ennemi → appel du super() de la classe “Ennemis” → appel de checkSommet() → retour à la méthode randomMob());

1.3 Structures de données :

La structure de donnée la plus avancée à laquelle nous ayons eu recours est la HashMap. Cette structure, qui implémente la classe Java “Map”, permet de synchroniser des données deux grâce à des liaisons entre celles-ci. Toutefois, elle n’est pas ordonnée, comme pourrait l’être une ArrayList par exemple, il est donc essentiel d’avoir connaissance de l’élément de départ afin de pouvoir dérouler la liste à partir du début.
On l’utilise dans la classe BFS, elle permet de contenir les différents Sommets en les synchronisant par deux, chaque sommet étant synchronisé avec un autre via l'algorithme du BFS.

Au niveau modélisation, nous avons utilisé un tableau d’entier pour représenter la map, chaque entier représentant un sprite différent, ce tableau est de base généré par le logiciel “Tiled” pour prévisualiser la génération de la map. Il est ensuite copié/collé dans la classe Map, et la classe viewMap permet via une formule de rogner l’image du Tileset de sorte que seul la tuile correspondante est chargée. Chaque élément du jeu fait 32 pixels de haut sur 32 pixels de large.

Chaque ennemi a une image correspondante, qui est générée via la classe viewEnnemi et qui différencie chaque ennemi via la méthode “instance of”. Chaque ennemi a une coordonnée x et y et une coordonnée qui est de 31 pixels supérieurs en x et y, ce qui forme sa hitbox.

Chaque tourelle a aussi une image correspondante, la méthode “instance of” permet aussi de différencier les tourelles entre elles. Leurs positions sont représentées par des coordonnées x et y, et pour que leur emplacement sur la map soit cohérent avec les cases

du terrain nous avons appliqué une division par 32 pour avoir une valeur entière et une multiplication par 32 pour qu'on tombe sur un multiple de celui-ci.

1.4 Exception :

Nous avons eu recours à deux Exceptions dans notre code pour sécuriser son exécution.

- La première exception est dans la classe "Vague". Elle consiste à vérifier que les ennemis apparaissent bien sur un des chemins praticables, mais aussi sur un des sommets compris dans le BFS. Dans le cas contraire, on lui donne alors des coordonnées de spawn défini manuellement, ce qui empêche des erreurs lors de l'apparition des ennemis.
- La deuxième se situe dans la classe "viewMap". Elle s'active dans le cas où une des tuiles de la map aurait une valeur qui n'est pas associée à une des tuiles possibles pour l'affichage. Si elle s'active, l'Exception va remplacer la valeur de la map par une tuile choisie à l'avance, afin d'empêcher une erreur lors de la génération ou d'avoir une case sans tuile.

1.5 Utilisation maîtrisée d'algorithmes intéressants :

L'algorithme le plus intéressant du projet est celui du BFS, dont le fonctionnement se déroule principalement dans les classes "BFS", "Sommet" et "Graphe". Lorsque que le "Jeu", qui sert d'environnement pour les différentes entités, est créé, on va créer en parallèle un graphe qui va associer à toutes les cases praticables pour les ennemis un sommet. On va aussi repérer la case avec la valeur 100 qui va servir de point de départ au BFS. On va ensuite créer les arêtes du graphe entre chaque sommet adjacent. Ensuite l'algorithme va se lancer et créer une queue contenant les sommets, qui vont être associés les uns aux autres grâce à la Hashmap. Les ennemis vont ensuite pouvoir parcourir remonter la Hashmap de sommet en sommet pour arriver à la fin du parcours.

1.6 Junits :

Aucune de nos classes n'est couverte par des tests Junit, nous manquions de temps pour compléter certaines parties du projet que nous pensions prioritaire, par conséquent nous n'avons pas pris le temps d'en faire.

2 Documents pour Gestion de projet

2.1 Document utilisateur (8 points) :

Candy Crash est un jeu de Tower Defense dont l'objectif est de protéger votre base de l'attaque de l'armée légume grâce à votre armée de bonbons. Le jeu se place dans un futur apocalyptique où les bonbons et les légumes ont pris vie et se mènent un combat sans merci ayant pour but la destruction de l'autre groupe. Pour vous défendre de l'attaque ennemi, vous aurez accès à différents pièges et tourelles sucrées pour empêcher les ennemis d'atteindre votre base.

Vous avez à votre disposition 6 structures pour mettre à mal l'attaque ennemi : 3 tourelles et 3 pièges.

La première tourelle est relativement simple, elle est manoeuvré par un ours en gélatine qui va tirer des balles à tête chercheuse sur l'ennemi à une fréquence relativement rapide en infligeant des dégâts moyens. La seconde tourelle est dirigé par un caramel qui va réduire de moitié la vitesse des ennemis lorsqu'ils passent à proximité de l'endroit où la tourelle est posée. La troisième tourelle fonctionne comme un mortier, elle va envoyer une charge sur l'ennemi à un moment donné, sans essayer de prévoir les déplacements. La charge inflige de lourds dégâts en zone, par conséquent la fréquence de tirs a été réduite par rapport à la tourelle de base.

Les pièges ont pour particularité de se poser directement sur le chemin des ennemis et s'active lorsqu'un ennemi passe dessus. Le premier piège est un ressort qui va renvoyer l'ennemi à un des spawns de la carte choisi aléatoirement. Ce piège peut s'activer une fois par vague au maximum. Le second piège est une trappe qui, une fois activé, va rester ouverte pendant une durée assez courte. Tant qu'elle est ouverte, tous les ennemis passant dessus sont automatiquement tués, et ce peu importe la quantité de points de vie qu'ils leur restaient. Là encore, ce piège s'active une fois et se recharge entre chaque vague. Le dernier piège est une mine qui va infliger de lourds dégâts en zone au moment de son explosion. Ce piège est à usage unique mais à un coût plus faible que les deux autres pour compenser ce désavantage.

Le joueur affrontera 4 types d'ennemis différents. Le premier est la Carambo, un ennemi plutôt rapide avec des points de vie moyens, qui n'a pas de mécanique particulière. Le second est la Gladiem, une pomme de terre assez lente qui a comme particularité de désactiver temporairement les tourelles autour d'elle lorsqu'elle meurt. Ses points de vie sont légèrement supérieurs à ceux de la Carambo. Le troisième ennemi est la Plank, un tank robuste mais plutôt lent. Il sert de planque au quatrième ennemi, le Pépino, un ennemi rapide mais fragile, sans mécanique particulière. Quand la Plank meurt, elle récupère tous ses points de vie et devient immobile. Dans ce mode, elle sert de point d'apparition pour les Pepino qui vont apparaître régulièrement jusqu'à ce que la Plank soit détruite à nouveau. Il est important de noter que les Pépinos ne peuvent apparaître qu'à travers la Plank et non pas à travers les spawns propre aux différentes cartes. Une autre chose que le joueur pourra remarquer est que la vitesse du Pepino est égale à celle des

balles de la tourelle de base, par conséquent il est très compliqué de tuer un Pépino sans tourelle de ralentissement. Nous avons fait ce choix pour forcer le joueur à réfléchir à ses achats et ne pas se contenter d'acheter des tourelles de base en grande quantité.

L'avancée du joueur n'est marqué que par le nombre de vagues, aucun système de score n'a été développé. Pour gagner le niveau, le joueur doit venir à bout de la 5e vague d'ennemi sans que sa base tombe à 0 point de vie.

Le joueur a évidemment la possibilité d'acheter des tourelles et des pièges quand il le désire tant qu'il a en sa possession la quantité de ressources nécessaires, ressources récupérables en tuant des ennemis. Il peut aussi vendre les objets qu'il a posé afin de récupérer des ressources, toutefois chaque vente ne lui rapportera que la moitié de la somme dépensé lors de l'achat.