Rapport de conception du projet de programmation orientée objet

Licence d'informatique – 2ème année
Faculté des sciences et techniques de Nantes

Galactic Shooter

présenté par
IJJA Ziad, LE FUR Victorien, PARADIS Alexandre, TABTI Rayane
le 24/11/2023

encadré par
Laurent Grandvilliers

Nantes Université

1 Cahier des charges

Galactic Shooter est un jeu de tir à la vue du dessus en 2D réalisé en Java grâce à l'interface graphique JavaFX. Ce jeu vidéo utilisera les différents concepts de la **programmation orientée objet (POO)** tels que **l'héritage, le polymorphisme, l'encapsulation**, etc... , pour parvenir aux différents besoins de l'utilisateur afin de terminer le jeu.

Le principe du jeu est d'affronter de redoutables vagues d'ennemis pour accumuler des **points** et des **bonus**. Après avoir vaincu toutes les vagues d'ennemis, le **boss** fait son apparition et un grand affrontement fait face entre le joueur et le boss. Une fois le boss vaincu, ou bien une fois ayant perdu, la partie se termine, affichant le **score** du joueur et le **plus haut score** enregistré.

Le joueur contrôlera un **vaisseau spatial**, pouvant se déplacer que de **droite à gauche**, et peut tirer vers le haut en direction de ses ennemis. Les ennemis quant à eux, auront plus de libertés de mouvements et auront des chorégraphies de mouvements différents en fonction du type de l'ennemi. Les commandes du joueur sont **FLECHE GAUCHE** pour se mouvoir à gauche, et **FLECHE DROITE** pour se mouvoir à droite. Enfin, il utilisera la **barre d'espace** pour tirer sur les ennemis.

Un **menu principal** sera présent dans le jeu pour effectuer différentes actions telles que **démarrer une partie**, **quitter le jeu**, ou **aller dans la boutique**. L'utilisateur pourra y user de ses pièces obtenues en combattant des ennemis pour acheter des **customisations à son vaisseau** (changement de couleur).

En somme, **Galactic Shooter** est un jeu d'arcade **dynamique** et **intense**, nécessitant **de bons réflexes** pour esquiver les nombreuses menaces que présentent les ennemis. Les joueurs s'affronteront ainsi pour obtenir le **meilleur score** et figurer parmi les **meilleurs joueurs**.

2 Architecture

2.1 Description générale

La classe **Entity** constitue la base fondamentale de la structure du jeu Galactic Shooter. Elle est conçue comme une classe abstraite, fournissant une base commune pour tous les éléments du jeu tel que la représentation des vaisseaux spatiaux, les ennemis, les objets collectionnables et les balles. Cette classe permet une gestion et une utilisation flexible et modulaire pour ses sous-classes.

La classe **SpaceEntity** est une classe abstraite représentant une entité spatiale générique dans le jeu. Elle est une sous-classe de la classe **Entity** et fournit des propriétés et des méthodes communes à toutes les entités spatiales, telles que la position, les points de vie, la gestion de collision, ou l'ajout d'armes. La gestion des collisions interagis avec tous les types d'entités tels que les **BulletEntity**, les **CollectibleEntity**, et les **SpaceEntity**.

La classe **BulletEntity** est une classe abstraite représentant un projectile générique dans le jeu. Elle est une sous-classe de la classe abstraite **Entity** et fournit des propriétés et des méthodes communes à toutes les entités de type **BulletEntity** dans le jeu, telles que les dégats ou l'effets des balles.

La classe **Weapon** est une classe abstraite représentant une arme générique dans le jeu. Elle fournit des propriétées communes telles que une liste de balles émises, un délais de tir. Les méthodes incluent l'ajout et la suppression de l'utilisateur, la gestion des balles, la définition du délais, ainsi que la méthode abstraite de tir. Cette méthode est donc reliée à la classe **BulletEntity** ainsi que la classe **SpaceEntity**, et sert d'intermédiaire entre ces deux classes.

La classe **CollectibleEntity** est une classe abstraite représentant un objet générique ramassable par le joueur. Elle est une sous-classe de la classe abstraite **Entity** et fournit des propriétées communes telle qu'un score gagné lors de l'intéraction ainsi que des méthodes communes à toutes les entités de type **CollectibleEntity** tel que l'effet au contact avec le joueur ou le comportement en jeu.

La classe **Enemy** est une classe représentant une entitée ennemie qui cherchera à détruire le joueur. Elle est une sous-classe de la classe abstraite **SpaceEntity** et fournit des propriétés communes telle que le score obtenu par le joueur à la destruction ou encore le comportement aléatoire ou non ainsi que des implémentations de méthodes communes à toutes les entitées de type **Enemy**.

La classe **Player** est une classe représentant le joueur. Elle est une sous-classe de la classe abstraite **SpaceEntity** et fournit des propriétées propre à cette classe relatif

à son mouvement ou à l'affichage de texte près de lui. Les méthodes lui étant propres servent à interpréter les touches saisies par l'utilisateur ainsi que toute les méthodes abstraites non-implémentées par **SpaceEntity**.

La classe **ViewManager** est une classe représentant la gestion de la logique interne du jeu ainsi que celle de l'affichage. Elle contient les différentes scnènes de jeu présentées au joueur. Elle contient donc les propriétées nécessaires à cela et propres à la bibliotèque JAVAFX. C'est dans cette classe que la boucle infini, permettant le déroulement du jeu, se situe.

La classe **SpaceInvaderGame** est l'Entry point, c'est la classe lancée à l'execution du programme. Elle est une sous-classe de la classe de JAVAFX **Application**. Elle contient une instance de la classe **ViewManager**.

La classe **Global** est une classe regroupant des propriétés accessibles à tout endroit du programme tel que la taille de l'écran de l'utilisateur, son score, des listes contenants les instances de la classe **Entity** ainsi que certaines méthodes qui sont utiles dans plusieurs classes indépendantes telles qu'une pour récupérer l'objet représentant le joueur.

La classe **TimedWave** est une classe représentant une liste d'ennemis qui doit arriver dans la vague actuelle après un certain temps. Elle contient des propriétés telles qu'une liste d'instances de la classe **Enemy**, le temps qu'elle doit attendre et une horloge pour le mesurer. Les méthodes qu'elle implémente servent à savoir si les ennemies doivent entrer dans le jeu.

La classe **clock** est une classe représentant une horloge. Elle contient des propriétés permettant de connaitre l'instant où elle a commencé, le temps qu'elle doit durer ou encore un objet JAVA permettant d'accéder au temps. Les méthodes qu'elle implémente permettent de la lancer, de mesurer le temps passé depuis son lancement et de savoir si elle a fait un "tour de son cadran" c'est-à-dire si il s'est écoulé le temps spécifié lors de sa création. Elle est utilisée pour fixer un minimum de temps entre 2 utilisations de l'arme par une instance de **SpaceEntity** ou encore dans la classe **TimedWave** pour savoir quand faire entrer la vague.

La classe **WaveParser** est une classe représentant un traitement de texte. Elle est utilisée pour lire le fichier "WaveData.txt" contenant les instructions pour la création des différentes vagues d'ennemies. Elle possède des propriétés telles qu'un objet de JAVA permettant de lire des fichiers ainsi qu'un booléen servant à marquer la fin du fichier. Ses méthodes permettent de lire le fichier ligne par ligne et vague par vagues en respectant une certaine syntaxe ainsi que la création d'ennemies en conséquence. C'est cette classe qui créera les instances de la classe **TimedWave** en fonction des spécifications du fichier. Elle est utilisée par la classe **ViewManager** uniquement.

La classe **WriteInFile** est une classe représentant un créateur de fichier texte. Elle est notamment utilisée pour créer des fichiers log ou encore enregistrer les meilleurs scores obtenus par l'utilisateur. Elle contient des propriétés lui permettant d'utiliser les classes de JAVA pour créer et accéder au fichier qu'elle édite. Ses méthodes lui donnent la possibilité de créer un fichier ou d'en remplacer un existant, mais aussi d'ajouter du texte dans un fichier existant, ce qui est utile pour sauvegarder de nouveaux scores sans effacer les anciens. Elle est notamment utilisée dans la classe **Global** pour pouvoir écrire dans le fichier log dans toutes les classes librement s'il y en a besoin.

La classe **Coin** est une classe représentant une pièce collectionnable par le joueur. Elle est issue de la super-classe **CollectibleEntity**. Elle est relâchée par les classes issues de **Enemy** et n'est obtenable que par **Player**. Elle contient des méthodes et attributs tels que sa valeur et son effet lors de sa collection.

La classe **HitBox** est une classe reliée à la classe **Entity** et représente la zone de contact de toute entité dans le jeu. Elle est représentée par un objet invisible **Rectangle** issu de JavaFX.

La classe **SpriteAnimation** est une classe reliée à **Entity** et sert à animer l'entité grâce à une liste d'images interchangées grâce à sa méthode **spriteAnimationPlay**.

La classe **GalacticGunner** est une sous-classe de **Enemy** et est l'un des ennemis présents dans le jeu. Il se déplace de gauche à droite en touchant les bords de l'écran, et tire de manière aléatoire. C'est l'ennemi de base du jeu.

La classe **CosmicCharger** est une sous-classe de **Enemy** et est l'un des ennemis présents dans le jeu. Il charge frénétiquement vers le joueur et est très fragile mais inflige de lourds dommages.

La classe **BasicWeapon** est une sous-classe de **Weapon** et représente l'arme par défaut du jeu. C'est pourquoi elle interagit avec le bullet **BasicBullet**

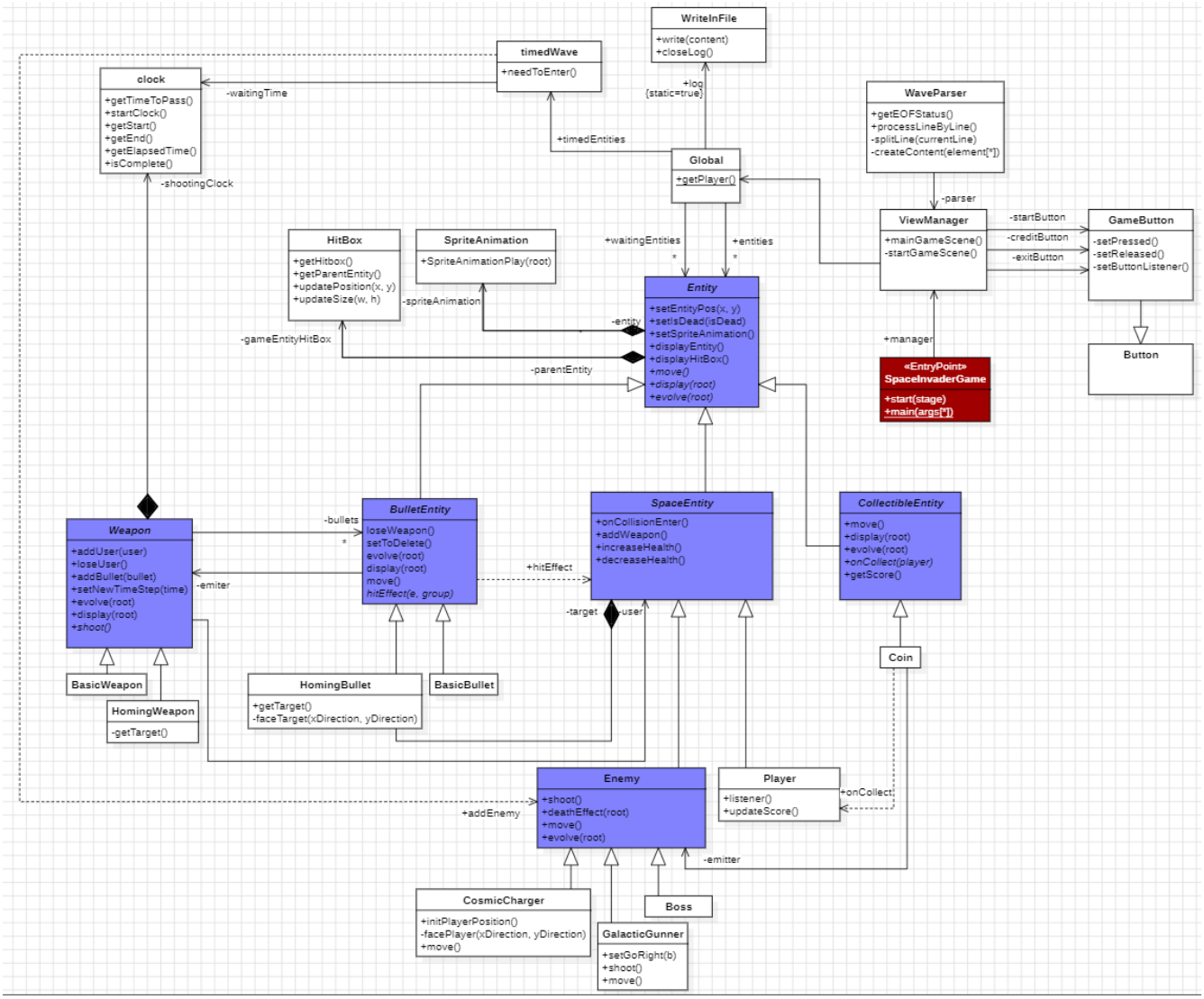
La classe **BasicBullet** est une classe représentant les balles classiques que nous tirons dans le jeu. Elle est une sous-classe de la classe abstraite **BulletEntity** dont elle override la méthode hitEffect.

La classe **HomingWeapon** est une classe qui représente une arme tirant un **HomingBullet** qu'elle crée lorsqu'elle est utilisée par un objet **SpaceEntity** qui est pris en paramètre par le constructeur. C'est une sous-classe de la super classe abstraite **Weapon** dont elle override la méthode **shoot**.

La classe **HomingBullet** est une sous-classe de la classe **BulletEntity**. Elle représente une balle qui poursuit la cible qui lui est assignée. Elle comprend de nombreuses méthodes telles que faceTarget qui permet à l'objet qui utilise cette méthode de faire face à sa cible, elle hérite de la super classe **BulletEntity** les méthodes **evolve** et **doesHit** qu'elle override.

La classe **GameButton** est une sous-classe de la superclasse **Button** , qui fait partie du package JavaFX. Elle permet d'instancier, dans la classe **ViewManager**, les boutons que nous avons dans le menu principal du jeu, lesquels permettent de lancer le jeu ou de le quitter. On utilise, pour représenter les objets de cette classe, deux images qui représentent le bouton pressé et relâché, ainsi que des méthodes permettant de naviguer entre les deux images lorsque le **ButtonListener** détecte que la souris clique sur le bouton.

2.2 Diagramme de classes



2.3 Interfaces

Lors du lancement du programme dans l'IDE, une première fenêtre s'affiche à l'écran représentant le menu du jeu, composée de plusieurs éléments tels que le logo du jeu, ainsi que les 3 boutons principaux : le bouton **START**, **QUIT** et **COMMANDS** (qui montre comment jouer). En cliquant sur le bouton **START**, ou en appuyant sur **ESPACE** ou **ENTRÉE**, la fenêtre du jeu principal s'affiche. La scène principale du jeu montre un vaisseau en bas de l'écran représentant le joueur, et en face, nous avons les ennemis. Nous avons également la barre de vie du joueur, ainsi qu'un bouton pause. Enfin, nous avons deux autres affichages : le score en haut à droite de l'écran, ainsi que le nombre de points gagnés par le joueur à chaque ennemi tué ou pièce récupérée. Chaque fois que l'on élimine tous les ennemis d'une vague, une nouvelle vague prend sa place. Après quelques vagues, le boss apparaît qu'il faudra battre, avant de voir s'afficher à l'écran l'écriture "**YOU WIN**" avec le score final ainsi que le plus haut score de toutes les parties jouées.

2.4 Aspects spécifiques

Galactic Shooter est un jeu complet, et à donc des algorithmes et des mécaniques complexes et sophistiquées, voici certains algorithmes importants :

- Les mouvements des **BulletEntity** utilisent des algorithmes de trigonmétrie afin d'ajuster continuellement sa direction vers sa cible, ces calculs sont essentiels pour modifier en temps réel l'orientation de l'image du projectile.

- Un algorithme permettant de retranscrire un fichier texte en vagues d'ennemis a été créé (**WaveParser.java** et **WaveData.txt** permettant ainsi une grande facilité de création de vagues, sans avoir besoin de modifier le code source du jeu.

- Une **HitBox** a été entièrement développé à l'aide de l'objet **Rectangle** de JavaFX, permettant à chaque entité du jeu d'avoir une véritable boîte de collision et donc d'être intersectable avec les autres entités. Les Rectangles sont ainsi ajouté dans le **Group root** du jeu, et c'est depuis cet élément que nous traitons les collision.

- Un algorithme complexe de détection de collisions (**onCollisionEnter**) a été réalisé afin de détecter les nouvelles collisions entre chaque élément du jeu, et traitant ces collisions en fonction de la classe des objets intragissant entre eux.

3 Regard critique

Dans le jeu Galactic Shooter, nous avons utilisé les concepts fondamentaux de la programmation orientée objet (POO). L'héritage est utilisé constamment, notamment pour la création de toutes les sous-classes de **Entity**, mais aussi pour les sous-classes de **Weapon**. Le polymorphisme a aussi été implémenté, avec les différents types de tirs et mouvements des ennemis et du joueur. L'encapsulation est utilisée pour permettre un accès restreint aux attributs privés et donc l'utilisation d'une interface (**Setters** et **Getters**), permettant d'interagir de manière sure avec les différents attributs. Les interfaces n'ont néanmoins pas été utilisées car leurs implémentations n'étaient à nos yeux pas intéressantes dans notre idée du projet, nous avons préféré utiliser seulement des classes abstraites(**Entity**, **SpaceEntity**, **CollectibleEntity**, **BulletEntity** et **Weapon**).

Le projet a été très sérieusement abordé et étudié dès le premier TP lié au projet, les idées étaient déjà claires et le groupe lucide sur ses compétences et la direction que le projet devait prendre. Nous annoncions à chaque rendez-vous les prochaines taches que nous devions réaliser, et comment nous y prendrons-nous. Aussi, nous avons utilisé Discord pour communiquer sur nos avancés, nos commits sur git, ou pour s'entraider à développer certains algorithmes.

Nous avons constamment travaillait sur le jeu Galactic Shooter depuis le début du projet, et plus nous travaillions, plus les nouvelles idées nous venaient, nous poussant ainsi à travailler encore plus dessus. Malheureusement l'un de nos membres a été un peu plus en retraite et n'a pas pu fournir autant de travail à ce projet.

Dans ce projet, l'un des membres détenait des connaissances assez avancées en Java, ainsi il a pu nous aider au début du projet pour bien comprendre les fondements du Java et de la POO. Nous nous sommes donc tous entraidés pour être à niveau sur le projet et développer autant l'un que l'autre.

Avec le WaveData.txt et le WaveParser, il est très simple de créer des vagues d'en-

nemis. Ainsi, il pourrait être possible de facilement renouveler le jeu en y ajoutant de nouveaux ennemis différents. Nous pourrions aussi partir dans l'idée de créer différents niveaux, ou bien aussi de nouvelles armes.

Bien que nous ayons eu directement nos idées et nos méthodes de travail, le professeur encadrant a toujours été à l'écoute de nos interrogations et a su y répondre clairement.