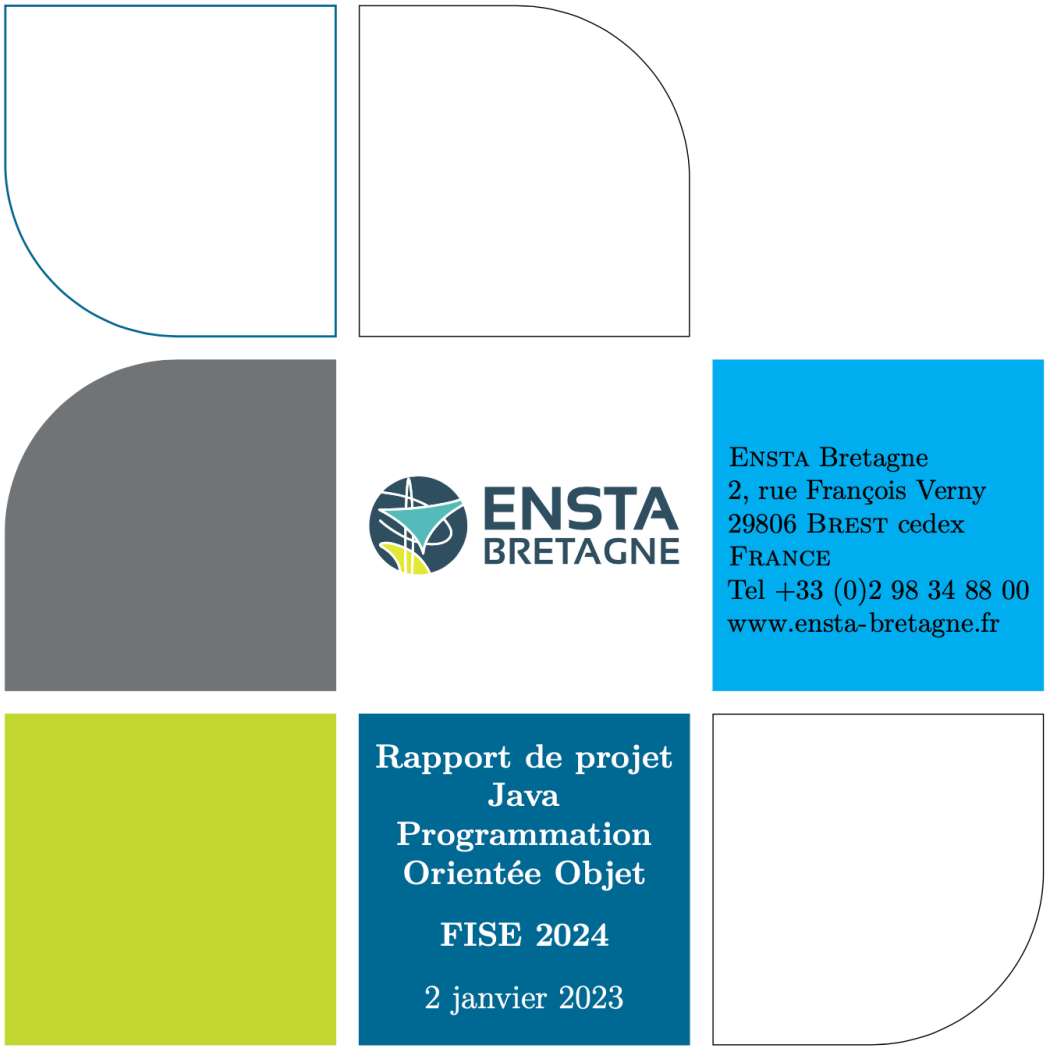

Antoine BREESÉ
FISE 2024 - CSN 2A
antoine.breese@ensta-bretagne.org

Henri LARDY
IETA 2024 – CSN 2A
henri.lardy@ensta-bretagne.org

Table des matières

Introduction

Ce projet s'inscrit dans le cadre du cours de programmation orientée objet en Java dispensé en deuxième année dans la filière Conception de Systèmes Numériques (CSN) à l'ENSTA Bretagne. L'objectif principal est de mettre en application les cours théoriques et les exemples d'applications réalisés en TD en développant un jeu vidéo en 2D qui fait appel aux notions travaillées. On s'attachera, au travers de ce rapport, à présenter le cahier des charges du projet en lui-même puis, dans une seconde partie, sa conception technique et l'IHM créée. La dernière partie présentera, quant à elle, un retour d'expérience sur la gestion du projet.

1. Présentation générale

a. Présentation du jeu à l'origine du projet

Le jeu vidéo sur lequel nous avons travaillé s'inspire du jeu d'arcade *Space Invaders* développé par *Taito Corporation* en 1978. Le succès mondial de ce dernier est à l'origine de nombreux jeux vidéos d'arcade en 2D.

Notre version reprend donc le principe de base : le joueur fait face à l'attaque d'une vague d'aliens qu'il doit tous éliminer pour gagner. Concrètement, ce dernier utilise un canon, mobile sur l'axe X, et peut tirer des projectiles en direction des aliens, selon l'axe Y. Ces derniers se rapprochent, quant à eux, du joueur en avançant en bloc selon l'axe -Y.

La version du projet adapte ce mode de jeu pour deux joueurs simultanés, chacun d'entre eux faisant face à une vague d'aliens. Pour remporter la partie, les aliens des deux joueurs doivent être éliminés. Il s'agit donc d'un « dédoublement du jeu d'origine en symétrie ».

b. Mécaniques de jeu et cahier des charges

i. Déplacements

Comme précisé ci-avant, le mouvement des différentes entités se fait :

- Canon (Player) : selon l'axe X, restreint à la fenêtre de jeu (impossibilité de sortir de cette dernière). La vitesse de déplacement est proportionnelle au taux de rafraichissement du jeu afin que son visuel soit le même quelle que soit la machine sur laquelle le jeu est exécuté.

- Alien : déplacement par bloc, selon l'axe Y et X (-Y et X pour le second bloc orienté vers le joueur2). L'armada se déplace donc horizontalement de gauche à droite et, lorsque l'un des aliens de l'armada atteint l'un des bords de la fenêtre du jeu, celle-ci se déplace d'une rangée vers le bas (respectivement vers le haut pour l'armada attaquant le joueur2)

- Blast : le tir du joueur se déplace verticalement selon l'axe X. Il est tiré depuis le canon du joueur à une vitesse définie.

ii. Collisions

Lorsque deux entités (Player, Alien ou Blast) entrent en collision, les mécaniques suivantes sont appliquées :

- Player vs Alien : mort du joueur qui conduit à la fin de partie, cette dernière étant alors perdue. Cette situation arrive lorsque la dernière rangée de l'armada arrive près de l'axe X et touche l'un des joueurs.

- Player vs Blast : mort du joueur qui conduit à la perte et fin de la partie. Cette situation arrive lorsque le tir de l'un des joueurs atteint l'autre joueur, sans avoir touché d'alien auparavant (sinon il est alors détruit).

- Blast vs Alien : élimination de l'alien et destruction du tir. Cette situation est très fréquente au cours de la partie, en effet pour remporter cette dernière il faut éliminer chacun des aliens des deux armadas.

iii. Conditions de fin de partie

Plusieurs conditions amènent à la fin de la partie :

- Victoire :
    - Les aliens des deux armadas ont été détruits sans qu'aucun des deux joueurs ne soit mort avant.

- Défaite :
    - Un alien d'une des deux armadas atteint la dernière rangée.
    - L'un des deux joueurs est touché par un alien.

c. Modifications apportées au cahier des charges

- Un délai a été ajouté entre deux tirs (Blast) par un joueur.

- En cas de collision entre les tirs de deux joueurs, les deux entités *Blast* s'éliminent mutuellement.

- Génération d'un fichier de logs à la fin de la partie. Ce dernier comprend l'horaire de cette dernière ainsi que l'issue : échec/victoire, le temps de jeu ainsi que, en cas de défaite, le nombre d'aliens encore en vie. En outre, les logs sont automatiquement générées par FXGL y sont aussi ajoutées.

2. Présentation et vérifications techniques

a. Architecture du logiciel

Le fonctionnement technique du logiciel se décompose en plusieurs classes, ayant chacune une fonction spécifique.

- **Classe *Constants* :**

Cette classe contient l'ensemble des variables utilisées pour paramétrer le jeu telles que les dimensions de la fenêtre du jeu, le délai entre deux tirs consécutifs par un joueur, la probabilité de tir d'un alien, etc.

- **Classe *InvadersApp :***

Cette classe est la classe principale du logiciel. Elle contient l'ensemble des méthodes nécessaires à l'initialisation du jeu ainsi que la méthode main.

- *initSettings()* : initialisation des paramètres de l'application.
- *initInput()* : association des touches de contrôle aux méthodes : principalement utilisée pour le déplacement des joueurs et le tir du canon. Un mode (paramètre de l'application à changer lors de l'initialisation précédente) permet d'accéder à des fonctionnalités de développement.
- *initGame() :* initialisation du jeu : création des aliens à l'aide de la classe InvadersFactory, initialisation de l'IHM, et création des deux entités associées aux joueurs (haut et bas).
- *initGameVars() :* initialisation des variables du jeu : nombre d'aliens restants (-1 ⇔ partie non commencée), booléen d'état du jeu et timer de début.
- *initUi() :* initialisation du label de nombre d'aliens restants.
- *initPhysics()* : initialisation des gestionnaires de collisions.
- *onUpdate()* : appelée à chaque frame, cette méthode gère la mise à jour de l'état de jeu.
- *endGame()* : méthode appelée par *onUpdate()* lorsque le jeu s'arrête. Elle affiche le message de fin de partie associé l'issue de cette dernière, joue le son correspondant et ajoute le label correspondant aux logs de la partie.

- *resetGame()* : appelée lorsque l'utilisateur relance une partie, permet de réinitialiser les paramètres de jeu.

- *main()* : Lancement du jeu.

- **Classe *InvadersFactory* :**

Cette classe est utilisée pour générer chacune des trois types d'entités : *Player, Alien* et *Blast.* Pour chacune d'entre elles, une méthode (par exemple : *newPlayer)* fait appel à la méthode EntityBuilder de FXGL pour générer une nouvelle entité dans le jeu. C'est dans cette méthode que l'entité reçoit ses composants : mécaniques (capacité de mouvement, hitbox), physiques (positions dans le jeu) et graphiques (textures, dimension, etc.).

D'autres classes, relatives aux composants, existent également.

- **Classes *EntityType* et *Directions :***

Ces deux classes contiennent chacune une énumération utilisée par les autres classes (respectivement pour les 3 types d'entités et pour les 4 directions de mouvement possibles).

- **Classes AnimationComponent, *PlayerComponent*, *AlienComponent* et *BlasterComponent :***

Ces trois dernières classes héritent de la classe *Component (*la classe *AnimationComponent* héritant elle-même de la classe *Component*) et gèrent les méthodes nécessaires au fonctionnement graphique des entités.

- **Classe *MoveComponent :***

Cette classe hérite de la classe *Component* de FXGL et implémente les méthodes liées au déplacement des entités (gauche, droite, haut et bas). Ces dernières sont appelées dans les autres classes selon les mouvements possibles décrits dans le cahier des charges du jeu.

- **Classe *AlienBlock* :**

Cette classe est un élément central du jeu car elle gère le déplacement des aliens en armada, détaillé dans la partie 1, et leur tir. Les méthodes principales sont résumées ci-après

- *AlienBlock() :* génère l'armada d'aliens selon les paramètres du jeu

- *findLeftMostAlien(), findRightMostAlien(), findTopMostAlien()* et *findBottomMostAlien()* : méthodes utilisées pour identifier la position des aliens « extrêmes », i.e. sur les côtés gauche et droit et, en haut et en bas.

- *removeAlien()* : appelée à l'élimination d'un alien, cette méthode entraine la suppression de l'entité du jeu, du dictionnaire des aliens et incrémente le compteur d'aliens tués.

- Les 4 méthodes de mouvement sont aussi redéfinies car le mouvement de l'entité *Alien* est lié à celui de l'armada et ne saurait être géré de la même manière que celui d'un *Player* ou d'un *Blast*.

Les diagrammes de chacune des classes présentées ci-dessus sont visibles en annexe.

b. Présentation de l'interface graphique

L'interface du jeu se décomposent en trois parties.

i. Menu principal et paramétrages des options

Comme visible sur la figure ci-dessous, le menu principal du jeu permet de lancer une nouvelle partie, de gérer les paramètres ou de quitter l'application.

L'image de droite présente, quant-à-elle, un exemple de sous menu de paramétrages. Ici, le menu permet d'associer les touches du clavier aux différentes actions de jeu.

Les menus étant automatiquement générés par FXGL, certains d'entre eux sont superflus (cas du sous menu Gameplay).

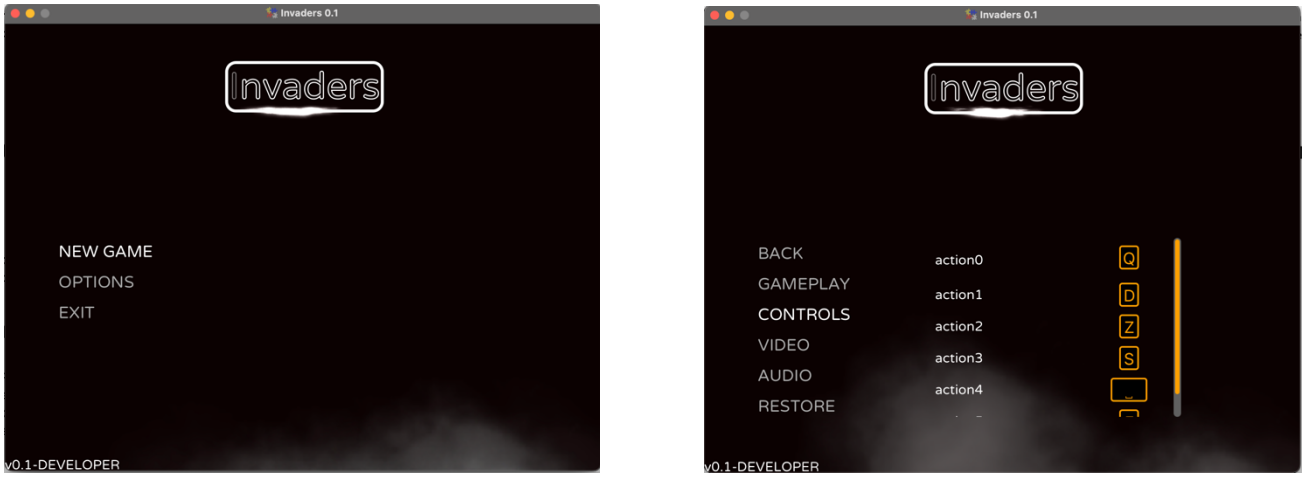

Figure 1 : Menus du jeu

ii. Interface en cours de partie

Une fois la partie lancée, la fenêtre suivante s'affiche. Celle-ci reprend les codes visuels du jeu à l'origine du projet. En début de partie, il y a donc, les deux joueurs (canons) centrés sur l'axe X1 et X2 (translation de X1 en haut de la fenêtre de jeu), ainsi que deux armadas (blocs) d'aliens orientées vers chacun des deux joueurs. Le nombre d'aliens restants est également affiché en haut à droite.

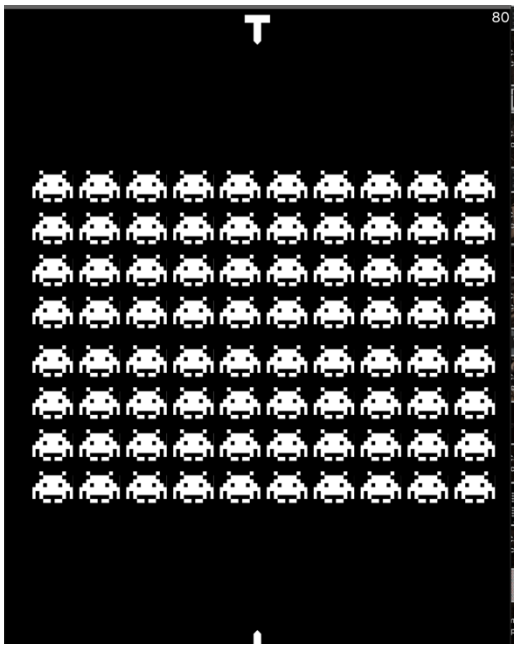

Figure 2 : Fenêtre du jeu en début de partie

Comme présenté précédemment, les aliens se déplacent horizontalement et verticalement en armada. Par ailleurs, ces derniers sont animés, leur texture est donc constituée de deux images (figure 3) qui se répètent en boucle.

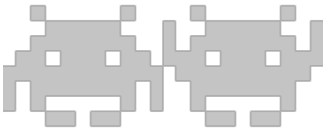

Figure 3 : Sprite d'un alien

iii. Fin de partie et logs

Selon l'issue de la partie, l'une des deux interfaces suivantes sera affichées. En cas de victoire, le temps de jeu est donné à titre indicatif (en vue de battre un précédent record par exemple). En cas de défaite, le nombre d'aliens restants est indiqué. En cliquant sur OK, il pourra ensuite relancer une autre partie.

Figure 4 : écrans de fin de partie, selon son issue

Par ailleurs, comme mentionné dans la première partie de ce rapport, des logs sont automatiquement générées. Un exemple de fichier log est visible en annexe du rapport.

c. Vérification du cahier des charges

Critère	Commentaire	Preuve/Référence
Code documenté	Les classes, méthodes et parties complexes du code sont documentées.	La documentation est visible sur la javadoc fournie en annexe.
Utilisation d'une bibliothèque graphique et justification du choix	Afin de faciliter le développement de l'IHM, nous avons fait le choix d'utiliser le framework de développement de jeu FXGL. Développé par Almas Baimagambetov, maitre de conférences et enseignant en sciences de l'informatique à l'université de Brighton, cette bibliothèque est particulièrement adaptée pour ce type de projets (jeux simples en 2D). En effet, elle permet de simplifier le code et de le factoriser rapidement. http://almasb.github.io/FXGLGames/	
Code D.R.Y (Do not Repeat Yourself)	L'utilisation du framework FXGL permet de faire de l'injection de dépendances, et d'utiliser de bonnes pratiques de la programmation orientée-objet, telles que les entités et les composants. En outre, nous avons aussi utilisé les héritages.	
Respect du principe de développement S.O.L.I.D[1]	Utilisation de composants, d'injection des dépendances, respect des conventions private/public…	
Tests unitaires	Un test unitaire a été réalisé pour tester la classe *MoveComponent* pour une entité *Player.*	
Respect des conventions de nommage	Le projet respecte les conventions de nommages propres à Java (packages en lowercase, classes en CamelCase, méthodes et variables en mixedCase, constantes en UPPERCASE) afin d'en faciliter sa lecture, sa maintenance et sa portabilité.	Exemple de nom de classe : public class InvadersApp Exemple de nom de méthode : protected void initInput()
Traitement des exceptions	Une partie des exceptions est automatiquement gérée par FXGL.	
Encapsulage des données	Les données sont bien encapsulées, par exemple, les constantes sont « public static final », et la plupart des variables et méthodes sont « private ».	

[1]Façon de concevoir l'architecture d'un programme informatique, afin d'en permettre une meilleure compréhension, extensibilité et maintenance, qui s'appuie sur 5 principes.

- Responsabilité unique : a une classe, fonction ou méthode ne correspond qu'une seule responsabilité
- Ouverture/Fermeture : le code doit être ouvert à l'extension mais pas à la modification directe
- Substitution de Liskov : la modification d'une instance dont dérive une autre ne doit pas nuire à la cohérence de cette dernière
- Ségrégation des interfaces : favoriser une interface spécifique à chaque type d'utilisateurs
- Inversion des dépendances : favoriser les abstractions aux implémentations

Portabilité du projet	Le code du logiciel ayant été réalisé dans le respect des conventions de nommage, et étant, de plus, documenté, la reprise ou la maintenance de ce dernier par un autre développeur est facilitée
Modules utilisés pour la généricité	Certains types de données génériques de la bibliothèque standard de Java ont été utilisés, tels que les HashMap ou les ArrayList.

3. Bilan sur la gestion du projet

Cette partie présente un bilan rapide sur la gestion générale du projet.

a. Logiciels et outils utilisés

Afin de collaborer à distance, nous avons utilisé les outils suivants :

- Un répertoire Gitlab organisé avec git-flow.
- Un dossier partagé sur Onedrive pour héberger différents documents et assets (sons, textures avant ajout sur le Gitlab) ainsi que le rapport écrit.

b. Bilan et améliorations possibles

Ce projet a été riche en enseignements et a permis de concrétiser les connaissances de cours au travers d'un jeu vidéo nécessitant une vraie réflexion sur la programmation orientée objet.

D'un point de vue technique, le temps alloué au projet n'a permis qu'une découverte assez sommaire de la bibliothèque FXGL. Au regard des nombreux tutoriels publiés par son créateur, de nombreuses autres fonctionnalités ou optimisation du code peuvent être apportées à notre version.

Sur le plan visuel, l'IHM est fonctionnelle et relativement esthétique. Un développement plus long aurait permis d'améliorer l'esthétique des différents éléments (Menus et entités). On peut par exemple renforcer le paramétrage du jeu au travers des menus d'options. Ces réglages sont en effet, dans notre version, fixés dans la classe *Constants*.

Sur le plan de la mécanique du jeu, de nombreuses fonctionnalités pourraient être ajoutées. De manière non exhaustive on peut imaginer celles suivantes :

- Ajouter plusieurs types d'aliens. L'élimination d'un certain type pourrait par exemple provoquer une explosion sur une distance définie.
- Ajouter un système de points de vie au joueur : actuellement un seul tir d'alien suffit à éliminer le joueur.
- Permettre à ce que le jeu continue après la mort du premier joueur, actuellement, l'élimination d'un seul des deux joueurs entraîne la perte de la partie.

- Mettre en place un système de niveaux de difficultés influant sur le nombre d'aliens initialement présents dans l'armada, leur vitesse de déplacement ou encore leur probabilité de tir.

Enfin, la gestion générale de ce projet fut une bonne expérience, bien que le faible niveau de connaissance de JavaFX ait entraîné une prise de retard significative avant le début réel de programmation du jeu.

Table des figures

Annexes

- Javadoc
- Diagramme de classe
- Exemple de fichier de logs