Projet Xpilot

2022

PI4

L2 Info

Groupe Y2J_B :

ZHOU Sebastien
HAMCHAOUI Ilyas
ZEYNALI Kyan
DAI Zhou
HUANG Longsheng

Sommaire

I – Introduction

Xpilot est notre premier grand projet en tant qu'apprentis développeurs. Ce projet représente un grand défi pour nous car c'est toujours particulier et loin d'être évident de concevoir tous les aspects d'un jeu en partant d'un dépôt vierge.
Notre objectif principal est resté le même, à savoir implémenter en java une version du jeu vidéo Xpilot ou Thrust fonctionnelle et stable, sans que le jeu soit pour autant magnifique visuellement ; avec cependant des ajouts et des modifications importantes pour rendre le jeu plus intéressant dans sa jouabilité et dans son ambiance.
En effet, les graphismes de notre jeu sont assez basiques, car les bibliothèques graphiques du Java Development Kit (JDK) sont très limitées comparé aux bibliothèques comme java FX, ou à des véritables moteurs graphiques comme Unity ou Unreal par exemple.

Au début du projet, lors de la conception initiale du jeu, on a dû faire de nombreux choix sur des éléments essentiels du jeu.
Dans un premier temps on n'avait pas un mais deux jeux sur lesquels on pouvait se baser : Thrust et Xpilot. Les deux sont cependant assez similaires. Par exemple au début, on avait choisi d'intégrer la gravité dans le jeu. On avait réussi à implémenter cette dernière, mais après une réunion importante lors de la deuxième semaine, on a décidé de la retirer.
En effet, on avait décidé que notre jeu se déroulerait dans l'espace avec le pilotage d'un vaisseau spatial. Par conséquent, ça aurait été incohérent de laisser la gravité alors que cette dernière n'existe pas dans l'espace.
Ainsi au final, on a décidé de créer un jeu de tir unique en intégrant des éléments provenant à la fois de Xpilot et Thrust.
Donc notre version comprend :

- Un vaisseau spatiale, qui n'est donc pas soumis à la gravité -> Xpilot
- Le contact avec les murs diminue considérablement la barre de vie du joueur -> Thrust
- Inertie du vaisseau en l'absence de gravité -> Xpilot
- Le joueur a un nombre de dégâts et de déplacements limités -> Thrust
- Boule à transporter, Point de départ, Point d'arrivé -> Thrust et Xpilot
- Ennemis, environnement hostile, obstacles -> Thrust et Xpilot
- Vue radar -> Xpilot

De plus nous avons ajouté les fonctionnalités suivantes :

- Murs cassables
- Menu principal et menu en cours de jeu
- Barre de vie, Barre de carburant, bonus pour récupérer du carburant
- Bouclier
- Multi missiles
- Les missiles tirés par le vaisseau peuvent rebondir sur les murs
- Musique spatiale / rétrofuturiste

Enfin, on a choisi le nom Xpilot pour notre projet car il nous était plus familier.

main

Le package « main » contient les classes qui forment la base de notre structure orientée objet. En effet, il contient les classes, les fonctions et les variables qui seront utilisées globalement dans tout le projet ; ainsi que la fonction main().

game

Le package « game » contient toutes les classes du jeu principal. On y retrouve la vue, le contrôleur et le modèle du jeu. Le modèle du jeu contient la fonction qui lance la boucle du jeu, qui est appelée dans le main().

object

Le package « object » contient les classes modélisant tous les objets non statiques du jeu, comme le vaisseau. En effet, il contient toutes les entités qui sont en mouvement dans le niveau.

map

Le package « map » contient les classes modélisant les objets statiques du niveau, comme les murs.
De plus, il contient les classes et les fonctions qui modélisent le design et le comportement général de la carte du niveau.

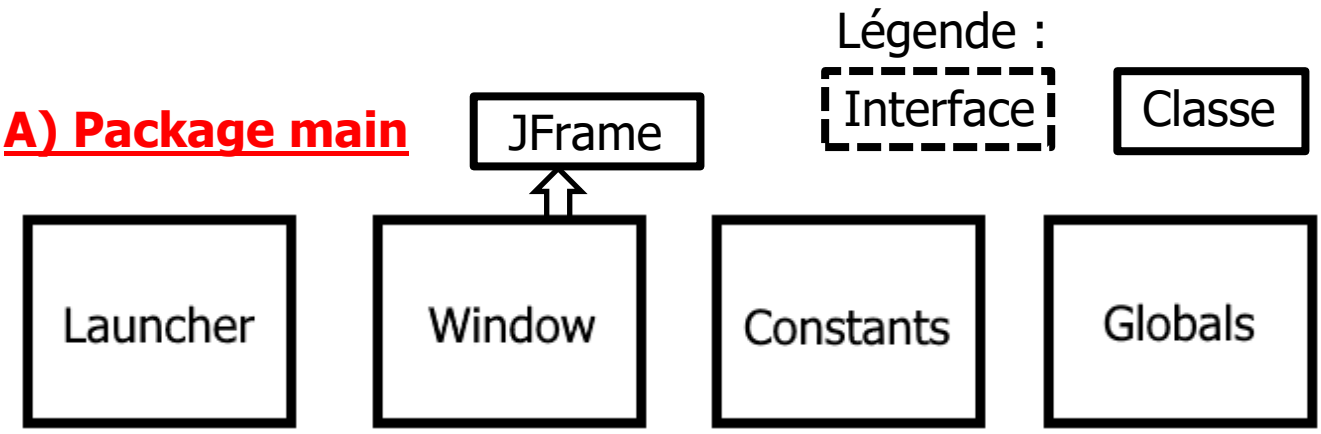
menu

Le package « menu » contient toutes les classes nécessaires au fonctionnement du menu principal et du menu en cours de jeu. C'est également dans ce package qu'on retrouve les éléments des sous-menus.

sound

Le package « sound » contient les classes qui modélisent les musiques et les effets sonores du jeu en objet. De plus, c'est dans ce package qu'on retrouve les fonctions qui gèrent les lecteurs audio ainsi que les volumes.

A) Package main

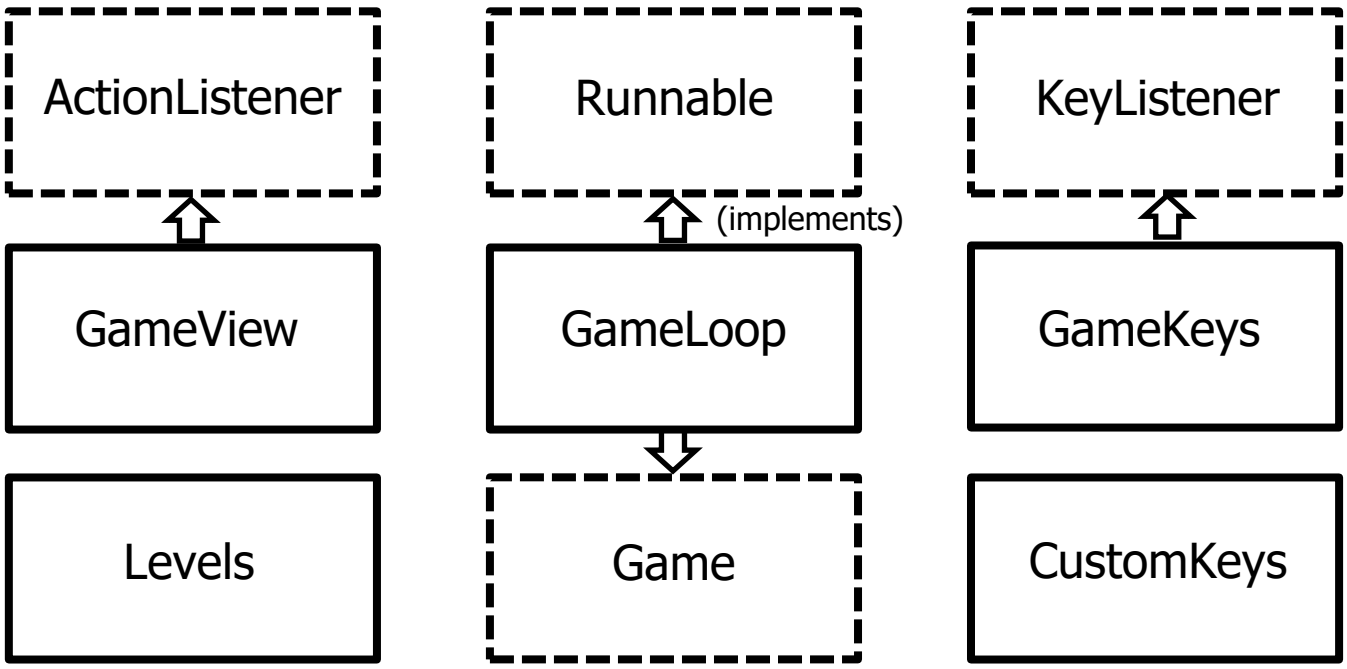


Tout d'abord, la classe « Launcher » contient la fonction main() qui lance l'exécution de l'application. Le main() va en fait lancer notre jeu en créant un nouvel objet de type « GameLoop ». Cette dernière est exécutable grâce à l'interface fonctionnelle « Runnable » de java et va donc être exécutée par le système d'exploitation grâce à la classe « Thread » de la librairie « System » de java. En effet, c'est dans le modèle du jeu, donc la classe « GameLoop » du package « game » où tous les paramètres propres au jeu sont initialisés.

Ensuite, c'est dans la classe « Window » où toutes les opérations sur la fenêtre principale sont gérées. Elle contient notamment les fonctions qui gèrent le mode plein écran, ainsi que du basculement entre le jeu principal et le menu. En effet, c'est dans cet objet qui est un enfant de « JFrame » qu'on va alterner entre les « JPanel » du jeu, du menu et ainsi de suite.

Les classes « Constants » et « Globals » contiennent toutes les deux les variables globales du projet. En effet, ce sont dans ces classes où sont stockées les objets et les valeurs importantes de l'application, et qui sont accessibles partout. La différence entre les deux s'explique sur le comportement de ces variables. Dans « Constants », sont stockées les variables constantes qui ne changent jamais de valeur (elles sont toutes en public static final). Au contraire, dans « Globals » les variables vont sûrement changer d'état pendant l'exécution du programme.

B) Package game



Le package « game » a pour objectif de stocker et de traiter les données du jeu principal, ainsi que les interactions avec l'utilisateur. Autrement dit, il rassemble le modèle dans « GameLoop » et « Game », le contrôleur dans « GameKeys » et « CustomKeys » et enfin la vue de l'utilisateur dans « GameView ».

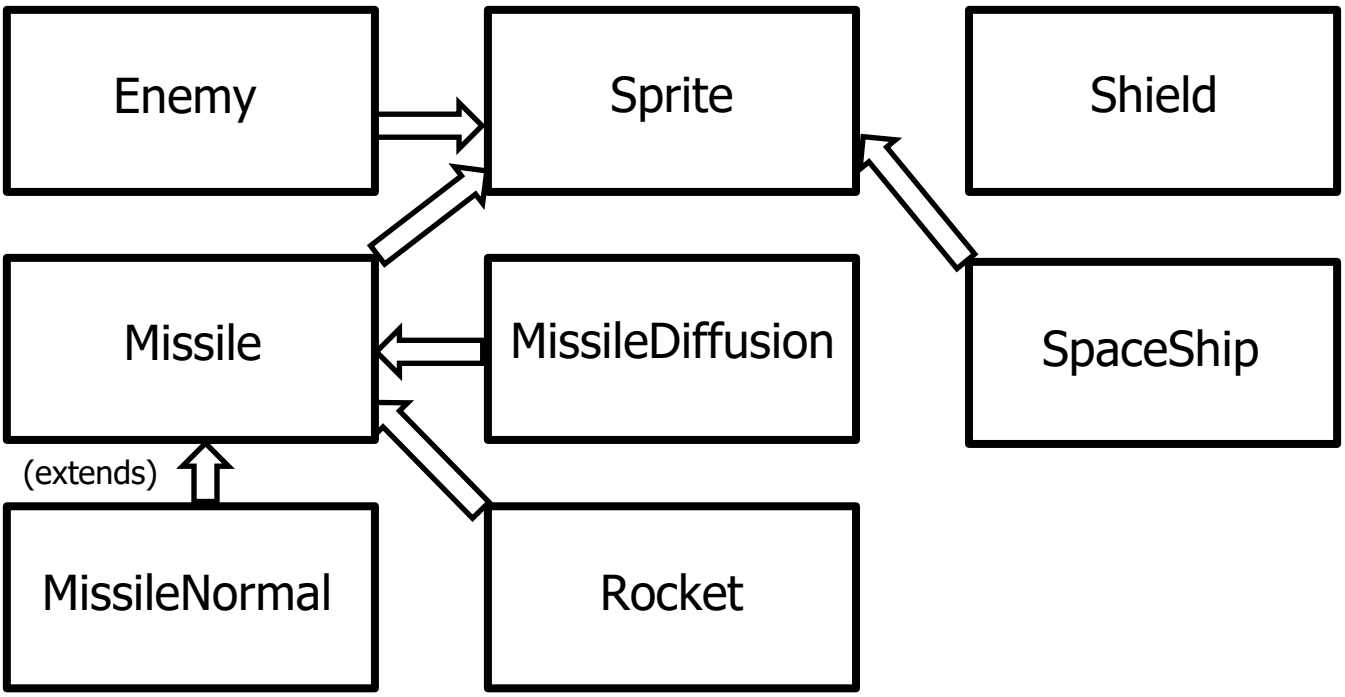
La classe « Gameloop » représente le modèle. Cette classe a pour rôle de stocker les données et de les traiter par le biais des fonctions qui régissent le comportement des éléments qui composent le jeu au fil du temps. « Gameloop » implémente l'interface Runnable qui nous permet de définir un thread via la fonction hérité run(). Ce thread contiendra une boucle qui se chargera d'exécuter les différentes fonctions qui mettent à jour l'état du jeu. « Gameloop » implémente également l'interface « Game », celle-ci permet de transmettre à la classe « Gameview » les informations contenues dans le modèle.

GameKeys joue le rôle du contrôleur, elle modifie les données du modèle en fonction des interactions de l'utilisateur avec le clavier, et cela grâce à l'interface « KeyListener » qu'elle implémente. La classe « CustomKeys » permet de configurer un mode de contrôle annexe proposant des touches différentes. Le jeu se joue effectivement de base avec les touches flèches du clavier, mais le joueur peut activer le mode WASD pour utiliser les touches W, A et D pour contrôler le vaisseau. Le fait que ces touches soient utilisées universellement par les jeux sur PC peuvent rendre le joueur plus à l'aise.

La classe « GameView » est un « JPanel », qui a pour but d'afficher dans la fenêtre tous les objets visibles du jeu que « GameLoop » utilise. « GameView » dessine le vaisseau « SpaceShip », les ennemies « Enemy », le bouclier « Shield », la barre de vie, la barre de carburant, le radar, les bonus « Bonus » à chaque seconde selon un ordre précis.
Si le jeu n'est pas fini, « GameView » continue de rafraîchir la vue toutes les secondes. Cependant, si le jeu est terminé, « GameView » affiche «Game over».
Le principe générale de l'affichage donné par les fonctions de « GameView » est d'abord de dessiner et fixer le vaisseau au milieu de la fenêtre. Ensuite on va dessiner tous les objets dans le panel avec au fond l'image de la « Map » où le vaisseau se positionne. Enfin, chaque mouvement du vaisseau change l'image de la « Map » et un repaint() est donc enclenché.
Pour l'affichage du jeu, nous avons une barre de vie et une barre de carburant qui sont respectivement rouges et vertes en haut de la fenêtre. Nous avons également un indicateur pour montrer le nombre de balles restantes et le type de tir utilisé en haut à gauche de la fenêtre. Pour le bouclier, il y a un indicateur du nombre de bouclier restant en haut à droite de la fenêtre. Enfin nous avons un radar situé en bas à droite de la fenêtre.

Les niveaux sont implémentés dans des fichiers texte qu'on stocke dans un enum pour pouvoir ajouter des niveaux facilement. Lorsque l'on termine le niveau, une fonction qui change le niveau courant est appelée, ce qui permet d'arriver au niveau d'après.
Le changement de niveau s'effectue en remplaçant la carte courante par celle du niveau suivant et en replaçant le vaisseau au point de départ. Quand on termine tous les niveaux du jeu, on arrive sur un écran "Game Win" où on peut rejouer ou quitter le jeu.
Il y a actuellement 3 niveaux dans le jeu :
Le premier niveau est un niveau tutoriel pour découvrir les éléments du jeu.
Le deuxième et troisième niveaux sont les deux niveaux principaux du jeu.

C) Package object



Le package « object » contient les objets dynamiques du jeu. L'objet principal est « Sprite » qui gère les coordonnées et les dimensions de l'objet. Le vaisseau et les missiles sont des objets hérités de « Sprite ».

La classe « Missile » est une classe abstraite qui contient les fonctions que chaque type de missile doit contenir. Ce sont les objets principaux du jeu avec lesquels le joueur interagit directement. Les différents types de missiles héritent de la classe « Missile » où ils ont des propriétés différentes.

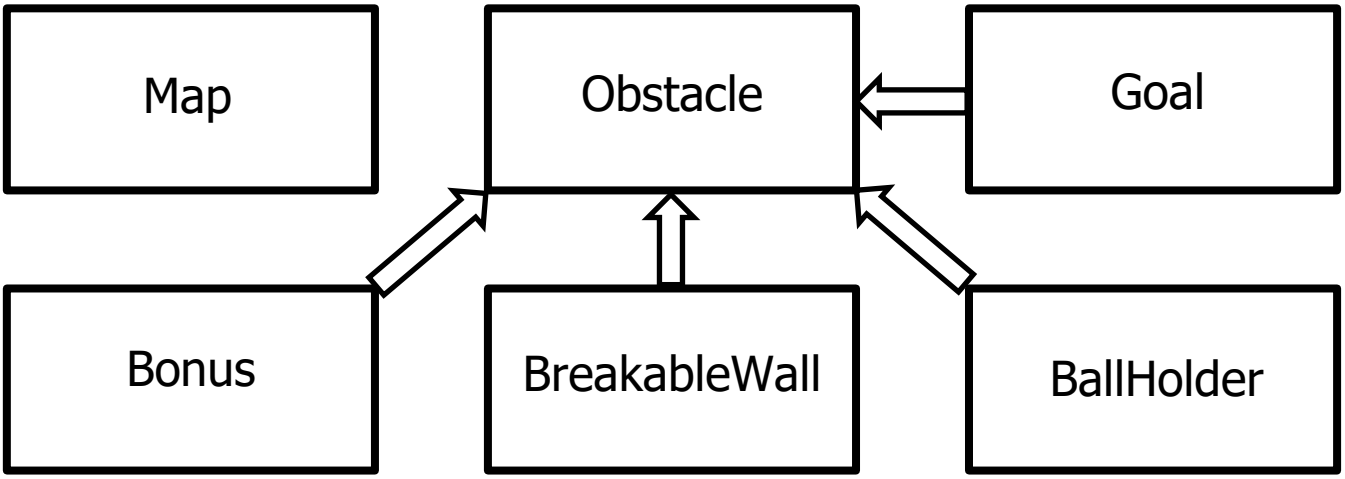
La classe « SpaceShip » est la classe la plus importante du package, elle contient toutes les fonctions et attributs gérant le vaisseau et le bouclier, c'est-à-dire la classe « Shield » du même package.

L'implémentation du vaisseau a été faite en plusieurs étapes. Au début on avait un vaisseau qui bougeait seulement dans la fenêtre du jeu. Puis nous avons implémenté le système de caméra qui montre seulement une partie de la carte dans la fenêtre. Donc visuellement le vaisseau est toujours au milieu de la fenêtre mais ses coordonnées changent lorsque ce dernier se déplace.

La classe « SpaceShip » héritée de la classe « Sprite » contient également toutes les fonctions qui agissent sur le vaisseau tels que la rotation et la vitesse.
La rotation est basée sur un système de 'flag', donc lorsqu'on reste appuyé sur la touche de rotation, l'action est activée et le vaisseau se met donc à tourner à certain taux (modifiable). Dès qu'on relâche la touche, la rotation se désactive.

A la première implémentation, la rotation était exprimée en degré mais pour faciliter le traitement de la rotation dans la vue, cette valeur sera finalement transformée en radian.
Le mouvement est une modification des coordonnées du vaisseau. Cette modification est calculée grâce à la vitesse du vaisseau et l'orientation du vaisseau.

D) Package map

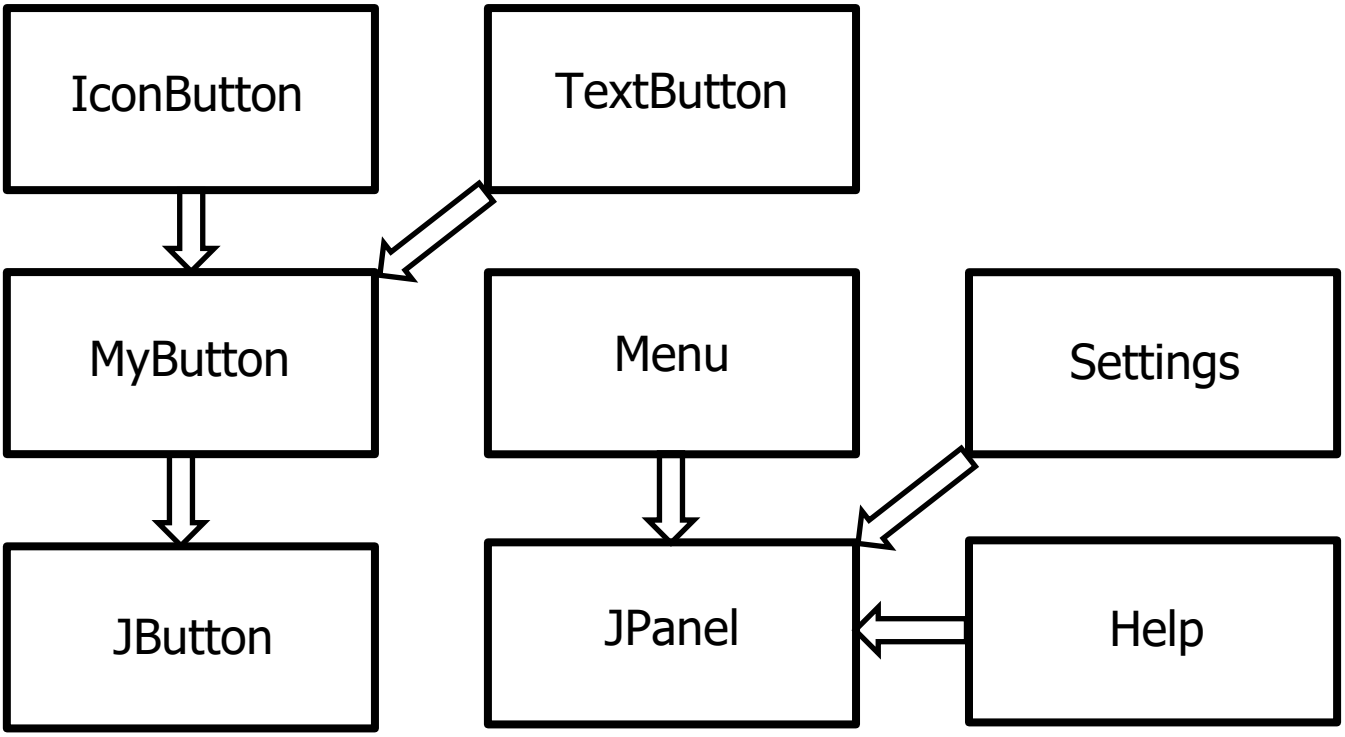


Dans un premier temps, la classe « Map » contient des objets comme « Enemy », « Obstacle », « Bonus », « Goal » et « BallHolder ». La création d'une « Map » est basée sur un tableau à deux dimensions d'une taille fixée. Ce dernier est défini grâce à une grille de caractères stocké dans un fichier texte dans le dossier « Ressources ». Map n'a pas de super classe.

La grille en question est constituée de plusieurs éléments, chaque objet visible étant en fait représenté par un caractère.
Tout d'abord, les murs sont représentés par 'W', ce sont les éléments principaux de la carte qui permettent de limiter les mouvements du joueur. Il y a aussi des murs cassables qui changent de couleur selon l'état du mur représentés par 'O', qui peuvent être détruit après trois tirs (ils passent du vert à l'orange puis au rouge). De plus il y a des boules de carburants qui apparaissent aléatoirement dans la partie libre de la carte qui est représenté par '-'.
De plus il y a des vaisseaux ennemis représentés par 'X' qui sont immobiles mais qui suivent de vue notre vaisseau et qui tirent lorsqu'on est à une certaine portée définie.
Enfin il y a l'emplacement de la boule et de l'objectif qui sont respectivement représentés par 'B' et 'G'.

Ensuite, on peut prendre les informations de cette grille et les stocker dans le tableau de la classe « Map » avec la méthode createinfor(String pathname) et puis on peut lire ces informations et créer le map en respectant le placement de chaque objet. Par conséquent on peut dessiner chaque objet de « Map » dans la position que l'on souhaite, car modifier le fichier texte revient à éditer le niveau dans son ensemble.

E) Package menu



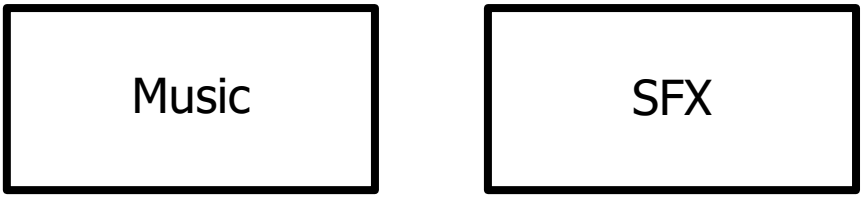
Tout comme la classe « GameView » du package « game », les classes « Menu », « Settings » et « Help » sont aussi des JPanel. Le menu principal est modélisé par la classe « Menu ». Selon l'état actuel de l'application, les boutons peuvent changer. En effet, si le joueur retourne au menu en cours de jeu grâce à la touche echap, le menu sera différent du menu de base.

Par ailleurs, on a décidé de donner une identité graphique au jeu, en créant des boutons spéciaux pour le menu. Pour cela on a créé notre propre bouton par le biais de la classe « MyButton » qui extends « JButton ». Cette classe abstraite permet la génération de deux types de boutons, les boutons avec un texte (classe « TextButton ») et les boutons avec une icône (classe « IconButton »).

Ensuite il y a les classes « Settings » et « Help » qui ont un comportement similaire à la classe « Menu » mais qui n'ont pas du tout les même fonctions. En effet, la classe « Settings » permet d'afficher le sous-menu où le joueur peut modifier certains paramètres du jeu.
Cette section propose le contrôle du volume en temps réel de la musique et des effets sonores par l'intermédiaire de curseurs (« JSlider »). De plus, on peut également activer ou désactiver le mode de commande WASD dans les paramètres (cf. package game).

Le détail des commandes du mode sélectionné est affiché dans le sous-menu help, grâce à la classe « Help ».

F) Package sound



Les classes « Music » et « SFX » transforment un chemin d'accès pointant vers un fichier audio encodé au format PCM (extension de fichier .wav) en un objet java.

Dans un premier temps, le constructeur va sampler le fichier audio pointé par le chemin passé en paramètre. Plus précisément, le fichier va être convertit en clip audio (classe « Clip »). Ensuite on va pouvoir gérer sa lecture et gérer son volume comme si on importait un logiciel de traitement audio, grâce aux classes et aux fonctions de l'api javax.sound.sampled.

La seule différence entre les deux classes va être leur utilisation dans le jeu, et leurs fonctions play() et stop() ne sont pas exactement les mêmes. La classe « Music » implémente un lecteur audio plus optimisé pour la musique, qui va par exemple reprendre du début lorsque la musique atteint sa fin. Le lecteur audio de la classe « SFX » est quant à lui plus optimisé pour les effets sonores, ainsi que pour les sons très courts.

III – Evolution

Dans cette partie, on va revenir sur les aspects principaux du jeu où on a rencontré des difficultés et diverses problèmes. L'intérêt de cette partie est de prendre du recul sur l'évolution de notre code. Il est également intéressant d'analyser les solutions trouvées à ces problèmes et comment elles ont été trouvées.

A) Caméra

L'aspect de la caméra qui suit le vaisseau fut une partie fondamentale dans l'implémentation de la vue du jeu. Pour cela, on a décidé de développer une vue en mode « verrouillage de l'objectif ». Effectivement, cela signifie qu'on veut que la caméra suive le mouvement du vaisseau, et le vaisseau que nous contrôlons est toujours au centre de la caméra.

Cependant, lors de nos premières implémentations, la caméra suivait bel et bien le mouvement du vaisseau mais ce dernier n'était pas au milieu de la vue. De plus on remarquait qu'à certains endroits précis du niveau, comme dans les quatre coins situés aux quatre extrémités, un décalage se produisait entre le caméra et le vaisseau.

Par conséquent, on a finalement opté une vue en mode « verrouillage de l'objectif » car elle fonctionne partout sans problème.

B) Collision

Nous avons écrit la méthode checkcollision() dans la classe « GameLoop » pour vérifier s'il y a un obstacle lors du mouvement du vaisseau et donc vérifier si ce dernier peut bouger. On avait besoin également de cette dernière pour implémenter les rebonds des tirs sur les murs. Le problème c'est qu'on ne savait vraiment pas comment implémenter cette méthode, car on ne trouvait pas de point de départ.
Pour remédier à cela, nous avons effectué de très nombreuses recherches. Au final, on a trouvé deux solutions pour vérifier un éventuel choc entre le vaisseau et un mur.

Pour la première solution, il fallait d'abord trouver les points représentant chaque côté d'une image visée, puis calculer la distance entre chaque point et le centre de vaisseau.
Si on avait (distance > rayon image vaisseau) alors il y avait un choc. Cette méthode est certes très précise, mais prend beaucoup temps à cause des nombreux calculs à effectuer.

Enfin, notre deuxième solution consistait à utiliser un classe de l'api de java qui permet la détection de collision de rectangles réguliers : x.getRect().intersects(x1.getRect()).
A contrario, cette dernière est efficace avec moins de calculs, mais elle manque de précision.
Au final, comme notre jeu ne contient pas d'objets vraiment complexes dans leur strcture, on a opté pour la deuxième solution.

C) Missiles

La principale difficulté rencontrée avec les tirs fut le calcul de la direction. Parfois on s'est mélangé les pinceaux et on a confondu les sens.
Pour remédier à cela, on affichait à chaque fois le direction obtenu, et la corrigeait au fur et à mesure que nos tests progressaient.

Ensuite on avait prévu de modéliser les missiles en un unique objet.
Cependant le comportement de la classe missile était trop simple.
On a donc ensuite décidé de développer les missiles avec plus de profondeur. En effet, on a créé une classe abstraite et développer deux sous-type de missiles différents qui ont chacun un comportement différent.

- La classe « MissileNormal » possède le même comportement que notre modélisation initiale des missiles, c'est-à-dire un seul missile avec une direction fixe.
- La classe « MissileDiffusion » modélise plusieurs missiles avec des directions différentes (entre direction (visée – angle) choisie et direction (visée + angle) choisie).

D) Séparation modèle / vue / contrôleur

Dans les premières versions de notre code, notre structure générale était assez brouillonne, car le modèle et la vue étaient tous les deux implémentés dans une seule et même classe (Board.java). Ainsi, un de nos premiers objectifs était donc de les séparés.
Pour remédier à cela, nous avons créé les classes « Gameloop » et « Gameview », la première chargée de contenir le modèle et la seconde la vue.
Cependant, la première version de ce changement n'était pas satisfaisante, et cela était dû au fait que les éléments du modèle étaient initialisés dans la classe qui contenait la vue.
Par conséquent, nous avons déplacé ces éléments dans la classe « Gameloop ».
De plus, cela nous a aussi amené à créer l'interface « Game » afin de transmettre les données du modèle à la vue.

E) Mode plein écran

Un autre de nos objectifs était d'implémenté un mode plein écran. Cette tache nous a obligé à modifier pas mal de choses notamment dans la vue ; car dans les versions antérieures, le radar était dans une classe à part qui extends « JPanel ». Par conséquent le redimensionnement de la fenêtre et des panels provoquait de nombreux conflits.
Pour palier à ce problème, nous avons donc intégré le radar directement dans la classe « Gameview », pour finalement n'avoir qu'un seul panel à afficher pour le jeu.

Nous avons également rencontré des problèmes d'affichage sur l'OS Windows, car le mode plein écran ne s'affichait pas au premier plan. On a résolu ce problème grâce à la fonction setundecorated(false).
De plus on a rencontré un autre problème sur les systèmes Linux. En effet, après le passage en mode plein écran il arrivait parfois que les inputs et les touches ne répondent plus.
Cela était provoqué par les fonctions du mode plein écran, car ces dernières étaient effectuées à partir de la boucle principale dans la classe « GameLoop » et cela engendrait de nombreux conflits ; donc nous les avons déplacés.

F) Vaisseau

Quelques difficultés ont été rencontrées durant l'implémentation de l'accélération et de la décélération du vaisseau. Au début nous avons essayé de voir comment ajuster notre modèle avec les formules de physique sur l'accélération et la décélération. Cependant, pour ne pas prendre le risque de corrompre le bon fonctionnement de notre modèle déjà implémenté depuis un certain moment, nous avons décidé de faire une version manuelle qui imite ce qu'aurait dû faire le vaisseau avec les formules de physique.
L'accélération a été implémenté en deux parties. Au début, l'accélération était linéaire mais puisque ce n'était pas très naturel, nous avons opté pour une accélération graduelle selon la durée de pression sur la touche. Plus on appuie sur la touche longtemps, plus on accélère. Evidemment, il y a une vitesse maximale qui peut être définie.

La décélération a été faite manuellement, lorsqu'on lâche la touche de mouvement, la vitesse diminue rapidement mais lorsqu'elle atteint une certaine vitesse, elle diminue très lentement, ce qui donne un effet d'inertie cohérent dans l'univers du jeu.

Le vaisseau possède des points de vie, un bouclier et du carburant. Tout d'abord, les points de vie sont représentés par une valeur numérique. Lorsqu'elle atteint 0, la partie est terminée. Ces points de vie peuvent être perdus soit à cause des ennemis, soit à cause d'un contact avec un mur. Le total de points de vie et les dégâts subis sont modifiables.

Ensuite, il y a le carburant qui permet au vaisseau de bouger, lorsqu'on en a plus, le vaisseau ne peut plus bouger, donc la partie se termine. Le carburant se consomme lorsqu'on appuie sur la touche de mouvement à une certaine vitesse et quantité (qui peuvent être modifiées). Du carburant peut être récupéré grâce aux boules qui apparaissent sur la carte.

Finalement, le bouclier est un objet utilisable en quantité limitée qui permet d'annuler la prise de dégâts lorsqu'il est activé. En effet, le bouclier est détruit lorsqu'on prend des dégâts.
On le reconnait grâce à l'hexagone autour du vaisseau. Ce bouclier peut être activer ou désactiver avec la touche associée dans les commandes.

Pour ces éléments il n'y a pas eu de grosses difficultés, mise à part la réduction du carburant par seconde, qui a nécessité un peu plus de réflexion, notamment à cause de la mauvaise organisation du modèle et de la vue lors de nos premières implémentations qui a rendu le travail un peu plus compliqué.

G) Menu

Au cours du semestre, notre système pour modéliser le menu a subi beaucoup d'évolutions. La première version du menu n'était pas du tout optimale, surement à cause du fait qu'elle ne possédait même pas de boutons cliquables. En effet, on devait avant sélectionner les section avec les touches flèches du clavier. De plus, le menu était basé sur un algorithme qui présentait des instabilités, notamment concernant la gestion de la frame.

Pour les sous-menu modélisés par les classes « Settings » et « Help », elles étaient de base des sous classes de menu. Cependant ce choix était incohérent, puisqu'elles ne pouvaient pas hériter de menu car leur contenu était différent.
Ainsi ce sont dorénavant deux classes distinctes, mais qui sont assez similaires dans leur fonctionnement et dans leur code.

Ensuite, il était grand temps d'ajouter de bons boutons pour naviguer en toute facilité dans les menus. Pour cela, nous avons créé nos propres boutons personnalisés dans une classe qui étend la classe « JButton » de l'api java. La difficulté a été de bien géré le contenu des écouteurs (action listeners) liés aux différents boutons.

Le problème majeur lors de la modélisation du menu était la gestion des panels. En effet, des recherches et des tests appondis ont dû être menées sur la disposition des panels. Au final, on a su faire face à ce problème en comprenant mieux le fonctionnement des fonctions setContentPane() et setLayout() de la classe « JPanel ».

Enfin, nous avons ajouté les fonctions qui permettent de placer les éléments d'interface graphique de tous les menus en position relative, car les UI components étaient à la base placés en codage dur, ce qui n'était pas beau visuellement lorsque le mode plein écran était activé.

H) Fonctionnalités abandonnées

Sont listées ci-dessous, les fonctionnalités qu'on a partiellement ou totalement implémentées à un moment dans notre code, mais qu'on a au final pas retenu dans notre version finale :

- boule qui est tenu par un fil

- personnalisation complète des commandes (cf. package game)

- gravité (cf. Intro)

- trainée du vaisseau

- système alternatif de collision (cf. Collision)

IV – Conclusion

Globalement, ce projet nous a permis de beaucoup progresser dans le domaine du développement, et ainsi mettre réellement en pratique tout ce que nous avons étudié durant ces 2 années de licence informatique générale.
De plus travailler sur un projet à 5 sur tout un semestre nous a permis d'améliorer notre sens de l'organisation et du travail d'équipe d'une part ; mais cela aura aussi surtout apporté un tout à chacun d'entre nous. Grâce à cette expérience, nous avons amélioré notre répartition des tâches ainsi que nos recherches, afin de trouver des idées pertinentes avec une certaine liberté, et s'entraider lorsqu'on rencontre des difficultés.
Aussi, nous avons à présent acquis un certain bagage technique programmation, notamment grâce aux divers objectifs que nous nous sommes fixés, mais aussi grâce aux problèmes auxquels nous avons été confrontés, qui nous ont poussé à enrichir nos connaissances.
Enfin, la conception de ce jeu nous a permis d'acquérir une expérience qui sera très enrichissante pour nos futurs projets.