LE ROUX Thomas
QUANTIN Thomas

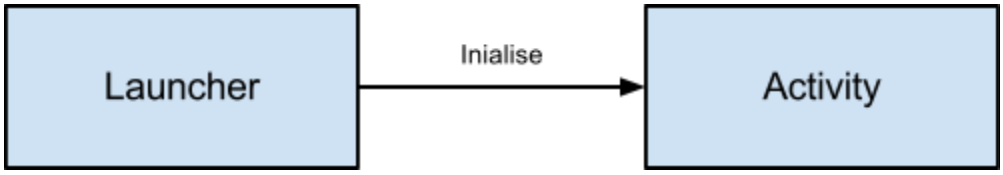
Escape - Architecture

Sommaire

I - Introduction

Nous allons vous décrire dans ce document l'architecture employée par notre jeu Escape.

Le lancement de ce jeu s'effectue en initialisant une **Activity** avec une **Configuration** et un **Game.**



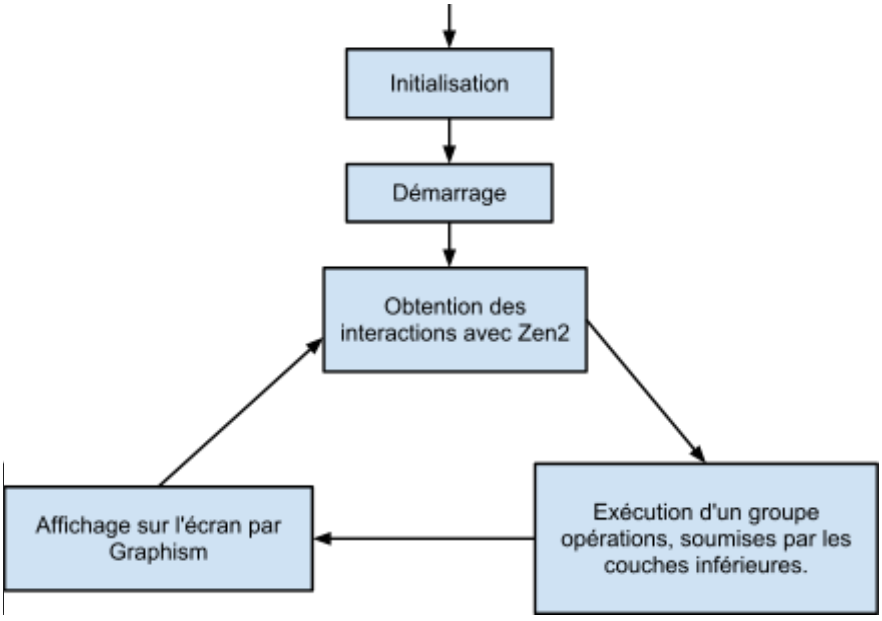
Une fois l'**Activity** initialisée et lancer, le jeu est opérationnel.
Nous allons vous expliquer comment fonctionne cette **Activity**.

II - Noyau du Jeu

Le noyau du jeu se situe dans le package *fr.escape.app.*
Ce noyau est composé de l'**Activity** et de **Graphism**.

1 - Activity

Activity le composant le plus important et critique dans notre application. C'est elle qui se charge d'initialiser les autres membres du noyau et les composants nécessaires pour **Foundation**.
Activity est composant cyclique ayant le cycle de vie suivant :



Ce cycle est effectué tant que l'utilisateur utilise notre application.

L'obtention des interactions sont effectuées par la bibliothèque Zen2. Ces événements sont ensuite propagés dans les couches inférieures par l'intermédiaire de **Game**.

Concernant la partie d'exécution de tâche, nous avons choisi pour cette première version d'effectuer ces opérations dans un environnement en *Single Threading*. Nous laissons la possibilité d'utiliser une Pool de Thread pour une version ultérieure lorsque notre application sera *Thread-Safe*.

Nous allons maintenant voir en détail le fonctionnement de **Graphism**.

2 - Graphism

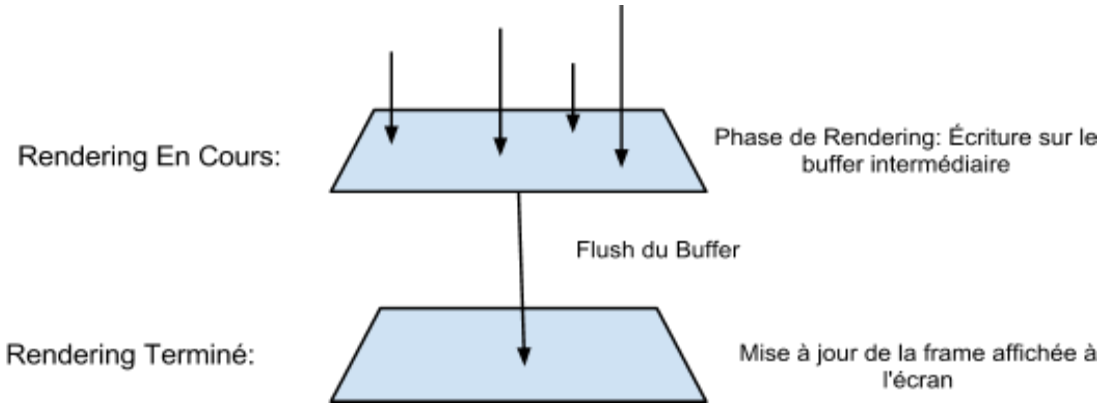
Graphism est un composant utilisé pour l'affichage utilisateur.

Elle s'occupe de charger et sauvegarder les informations fournies par l'utilisateur et son environnement, tel que le taux de rafraichissement, la dimension du jeu, etc...

Une fois cette initialisation effectuée, elle est utilisée par **Activity** comme sur-couche à awt et zen2, car elle simplifie chaque demande de rendering (draw) ...

Lors de la phase de rendering, elle calcule automatiquement son prochain réveil pour obtenir le taux de rafraichissement souhaité.

Enfin, elle utilise un buffer intermédiaire pour chaque phase de rendering pour minimiser l'utilisation de la carte graphique et obtenir un affichage beaucoup plus lisse, sans saut de frame.



Nous allons maintenant vous expliquer l'architecture utilisée pour la phase de rendering.

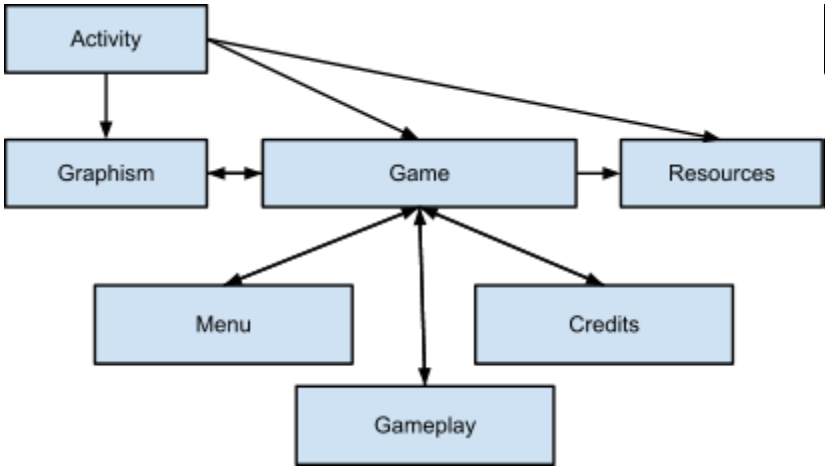
III - L'architecture Game et Screen

Game, dans notre architecture globale, est le contrôleur central. C'est celui qui reçoit des ordres des couches supérieures puis sous-traite ces tâches en utilisant les couches inférieures.

Game n'ayant pas le même comportement entre un écran d'accueil et le jeu du shoot'em up, nous avons introduit la notion de **Screen**.

Ce Screen, inspiré du design pattern Behavior, nous permet une gestion plus fine dans la sous-traitance des tâches.

En effet, Game utilise Screen pour effectuer de la délégation de rendering. Chaque Screen ayant un contexte et un état différent, elle demande à **Game** les composants nécessaires pour effectuer leur représentation :



IV - Entity et EntityContainer

1 - Entity

L'interface Entity nous permet de définir une base de méthodes communes à chacun des objets évoluant dans le jeu. Ces entités implémentant trois autres interfaces : **Collisionable**, **Moveable** et **Drawable**.

Ainsi, chacun des éléments composants le jeu (Ship, Bonus, Weapon) ont un squelette commun :

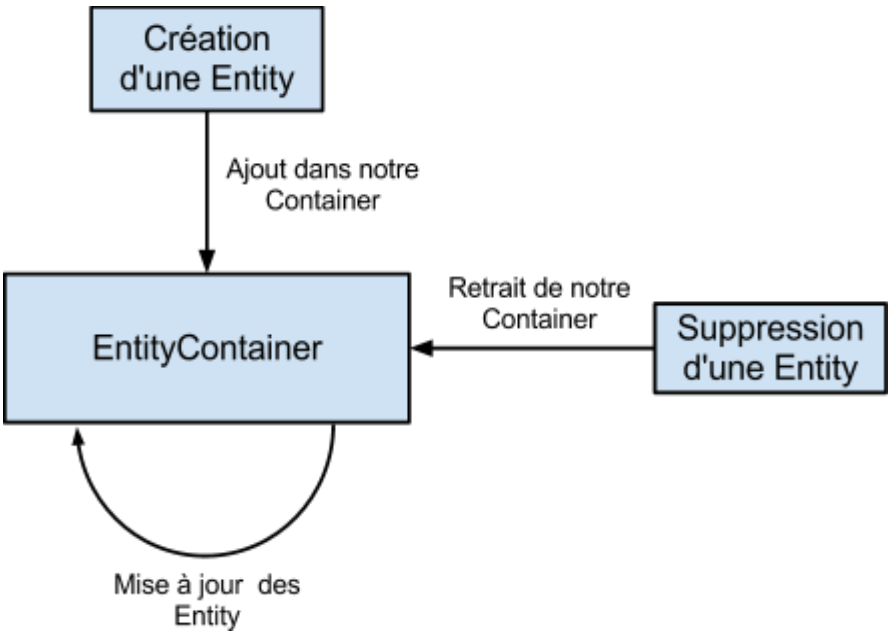
- L'interface **Moveable** permet de définir l'ensemble des méthodes permettant de déplacer un objet au sein du monde créée par JBox2D.
- L'interface **Drawable** contient tous les moyens permettant de dessiner un élément à l'aide de la classe Graphics vu précédemment.
- L'interface **Collisionable** fournit une unique méthode nous permettant de gérer les collisions des différentes Entity.

Cette base commune va nous permettre de facilité la gestion de l'ensemble des éléments du jeu, car nous n'avons plus qu'à traiter des Entity ayant des comportements similaires.

2 - EntityContainer

Une fois cette notion concue, il nous a fallu mettre en place un système nous permettant de gérer un ensemble d'*Entity* plutôt que de devoir les manipuler une à une. Pour cela, nous avons utilisé le Design Pattern *Composite*.

Ce choix nous a permis de créer un cycle sur nos entités existantes:



Notre container sera donc chargé de maintenir à jour la liste des éléments présents dans le monde, mais il devra aussi gérer leur suppression.

En effet, tout d'abord notre EntityContainer est chargé de vérifier que chacune des entités sont bien contenues au sein du monde et qu'elles ne sont pas sorties. Dans ce cas, l'EntityContainer doit la retirer de sa liste d'Entity active.

Mais il a aussi fallu faire en sorte de gérer correctement leur suppression. En effet, lorsque le monde est mis à jour une Entity ne peut pas être supprimé, il a donc fallu s'assurer que la mise à jour du monde et les suppressions ne s'effectuent pas au même moment.
Pour cela, notre EntityContainer maintient une liste temporaire des Entity à supprimer. Ainsi, elles ne sont plus afficher, car elles ont été retirées de notre liste principale, mais ne seront complètement détruites qu'à la fin de la mise à jour du monde JBox2D.

3 - Ship, Weapon et Bonus

1. Ship

Une fois notre système d'Entity / EntityContainer, nous avons pu nous attaquer à la création d'un vaisseau. Pour cela, nous avons créé une Interface Ship qui aura pour but de nous permettre de gérer les vaisseaux sans avoir à distinguer le joueur, des vaisseaux ennemis et des boss.

Nous avons ensuite pu constater que nombres des méthodes fournies par notre interface avait le même comportement quel que soit le type de vaisseau. Nous avons donc décidé de créer une classe AbstractShip, nous permettant de factoriser l'ensemble du code commun.

Au final, nous nous sommes retrouvé avec très peu de méthode à redéfinir par type de vaisseau (certains types ne redéfinissant rien). C'est alors posé la question de la création d'un nouveau vaisseau quel qu'il soit. Pour éviter de nous retrouver avec une classe par type de vaisseau, nous avons utilisé le Design Pattern Factory pour gérer cela.

2. Weapon

Le système d'armement est composé de deux éléments : l'arme (Weapon) et son projectile (Shot). Ainsi le Ship ne sera en interaction qu'avec la Weapon et jamais avec le Shot. Le Ship ne connaîtra donc que le moyen de tirer et de charger une arme. L'arme sera alors chargée de créer elle-même son projectile à l'aide de la ShotFactory.

Pour la gestion des Shot, nous avons créé une classe AbstractShot nous permettant de factorise l'ensemble des méthodes communes. Cependant, il restait un nombre important de méthodes propre à chaque Shot. En effet, en fonction de l'état du Shot (load, fire …), ils pouvaient présenter des différences importantes au niveau de leur comportement. C'est pourquoi nous avons définie une classe par Shot héritant de notre AbstractShot en plus d'une simple factory.

L'utilisation de la classe interne ShotContext de la classe Shot, nous permet d'avoir rapidement des informations sur le Shot. En effet, il nous était important de savoir, par exemple, si le Shot appartenait au joueur ou non, ou encore les dimensions de ce dernier pour gérer les collisions et son affichage correctement.

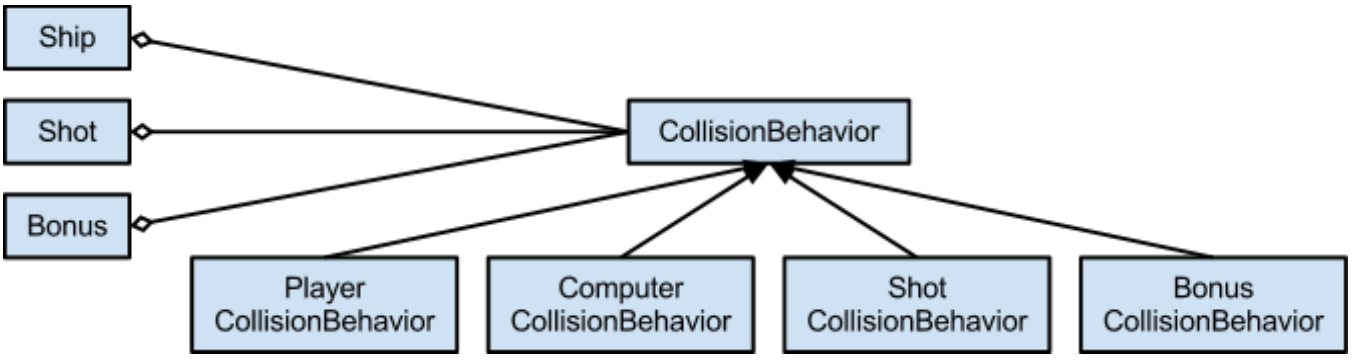
3. Bonus

Les bonus suivent le même principe que les Ship au niveau de l'architecture. La seule différence vient de la gestion de la création d'un Bonus. En effet, ils ne sont créés que lorsque un Ship est détruit. Nous avons donc délégué cette fonctionnalité à l'EntityContainer et au Ship qui lors de sa destruction indiquera à l'EntityContainer de faire éventuellement apparaitre un Bonus.

4 - Collisions

La gestion des collisions entre les Entity se fait en deux temps. Nous avons tout d'abord définie une interface **CollisionBehavior** qui nous permet d'ajouter à chaque Entity en type de comportement lors d'une Collision. Cette interface ne contient que la méthode permettant d'appliquer la collision.

Nous avons alors dû définir dans chacune des Entity un type comportement lors d'une collision :



Nous avons ensuite implémenter l'interface **ContactListener** fournit par JBox2D dans une classe que nous avons nommé **CollisionDetector** qui recevra lors d'une collision les deux Entity concernées.
Elle se chargera ensuite d'utiliser la méthode collision() de nos Entity pour pouvoir appliquer le traitement de la collision en fonction du **CollisionBehavior** que les Entity possèdent.

V - Niveau et Stratégie

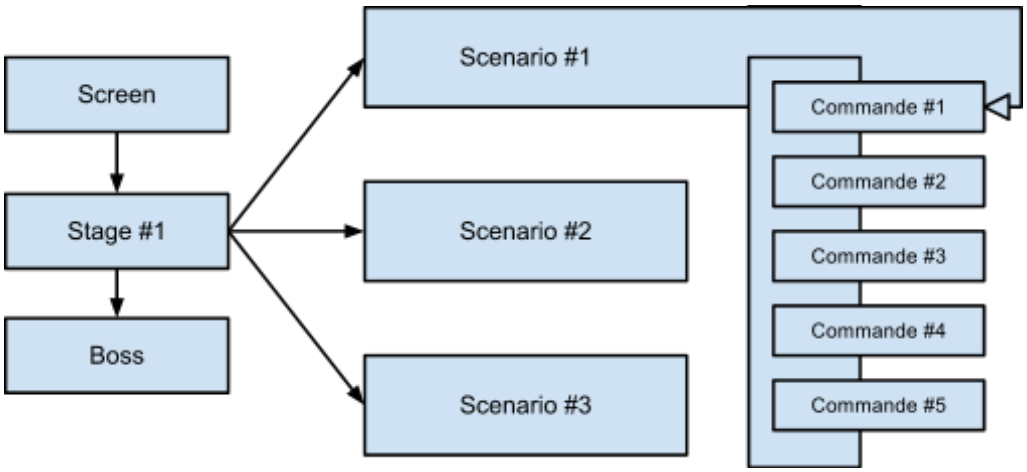
Nous avons trois niveaux dans notre jeu. Un niveau ayant plusieurs vagues d'ennemis, suivi d'un Boss pour passer au niveau suivant.

Nous avons donc découpé ce besoin en deux parties : **Stage** et **Scenario.**

- **Stage :** Représente un niveau constitué d'un ou plusieurs Scenario et d'un Boss.
- **Scenario :** Ensemble de commande pour les ennemis qui défini la stratégie globale de la vague.

La mise à jour du Stage est effectuée par la Screen qui la possède. Elle en profite par la même occasion pour vérifier si le niveau n'est pas terminé pour passer au suivant.

Le Stage se charge donc de switcher entre chaque *Scenario* actif / inactif en fonction de la progression globale du niveau, permettant ainsi de mettre à jour les Scenario actif pour en dérouler la stratégie globale.



VI - Conclusion

Nous avons énormement appris sur la culture des jeux vidéo ainsi que certains paradigme sur leur conception.

Ce projet fut sûrement l'un des plus intéressants pendant cette période. En effet, ce projet nous aura permis de mettre en pratique les différentes connaissances acquises en Java, mais aussi d'implémenter des Design Pattern.
Il fut très instructive sur le plan de la réflexion et de la conception d'un modèle de données qui devait respecter des contraintes claires et précises.

Enfin, ce fut une expérience enrichissante, beaucoup plus immersive que ce que nous avions eu jusqu'à maintenant.