

Dindane Samy
Faucillion Guillaume



Rapport

Sommaire

[Organisation du projet](#)

[Git](#)

[Trello](#)

[Boucle principale du jeu](#)

[L'architecture](#)

[Les formes et leur contact](#)

[Ennemis](#)

[Affichage et déroulement du jeu](#)

[Les entités du jeu](#)

[Vue d'ensemble](#)

[Les ennemis](#)

[Exemple : ajouter un nouveau type d'ennemi](#)

[Les deux ennemis supplémentaires](#)

[Les Bombes](#)

[Les planètes](#)

[Les déplacements](#)

[Les collisions](#)

[Le Radar](#)

[La configuration des niveaux](#)

[Exemple : ajout d'une nouvelle variable de configuration](#)

[Le Synchronizer](#)

[Le générateur de vagues d'ennemis](#)

Organisation du projet

Git

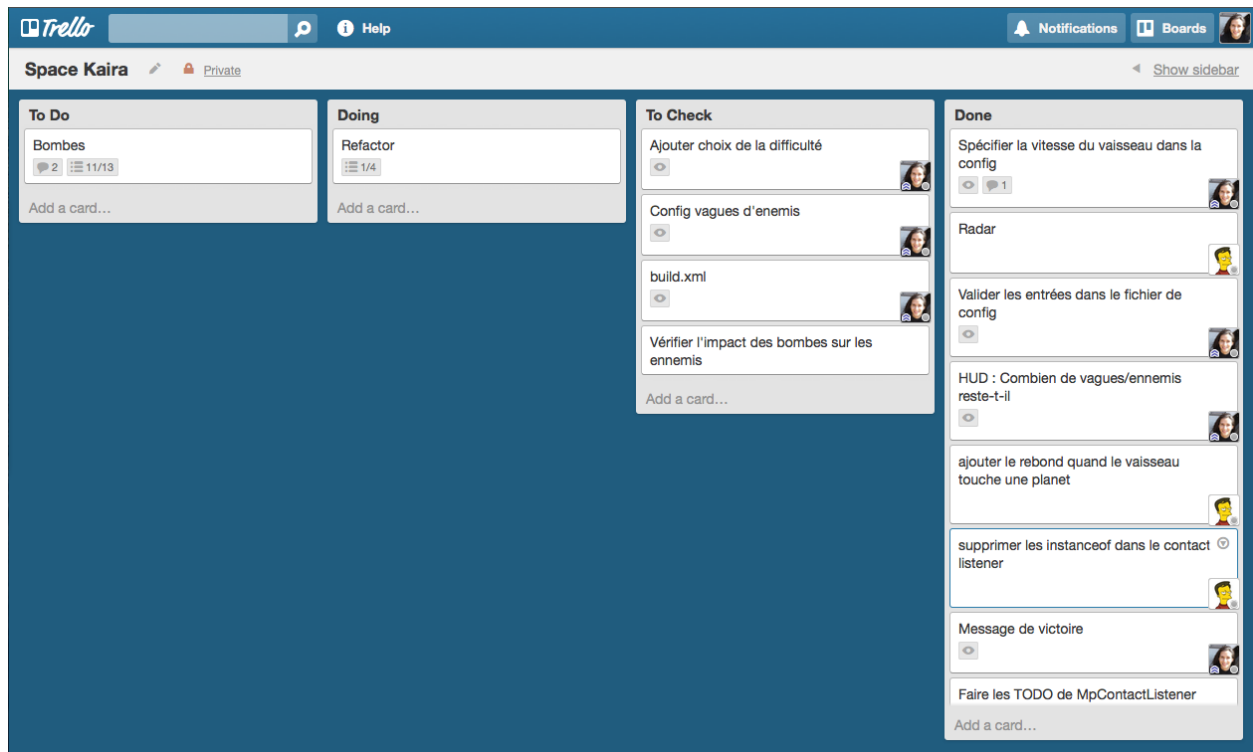
On a utilisé Git pour versionner le projet. Ça nous a permis de travailler ensemble sur la même base de code, d'implémenter différentes fonctionnalités sans se marcher sur les pieds, de revenir en arrière en cas de régression, etc.

Le projet est disponible sur GitHub sur <https://github.com/dinduks/space-kaira>

Trello

Trello un est outil en ligne qui offre la création de todo lists. Il nous a permis de noter les tâches à faire et d'avoir une vue globale sur ce qu'on fait et ce que fait notre binôme.

Il nous a aussi facilité l'écriture du rapport car on pouvait parcourir la liste de toutes les tâches faites précédemment.



Boucle principale du jeu

Une fois toutes les initialisations faites, le jeu se résume à une boucle infinie, qui attend :

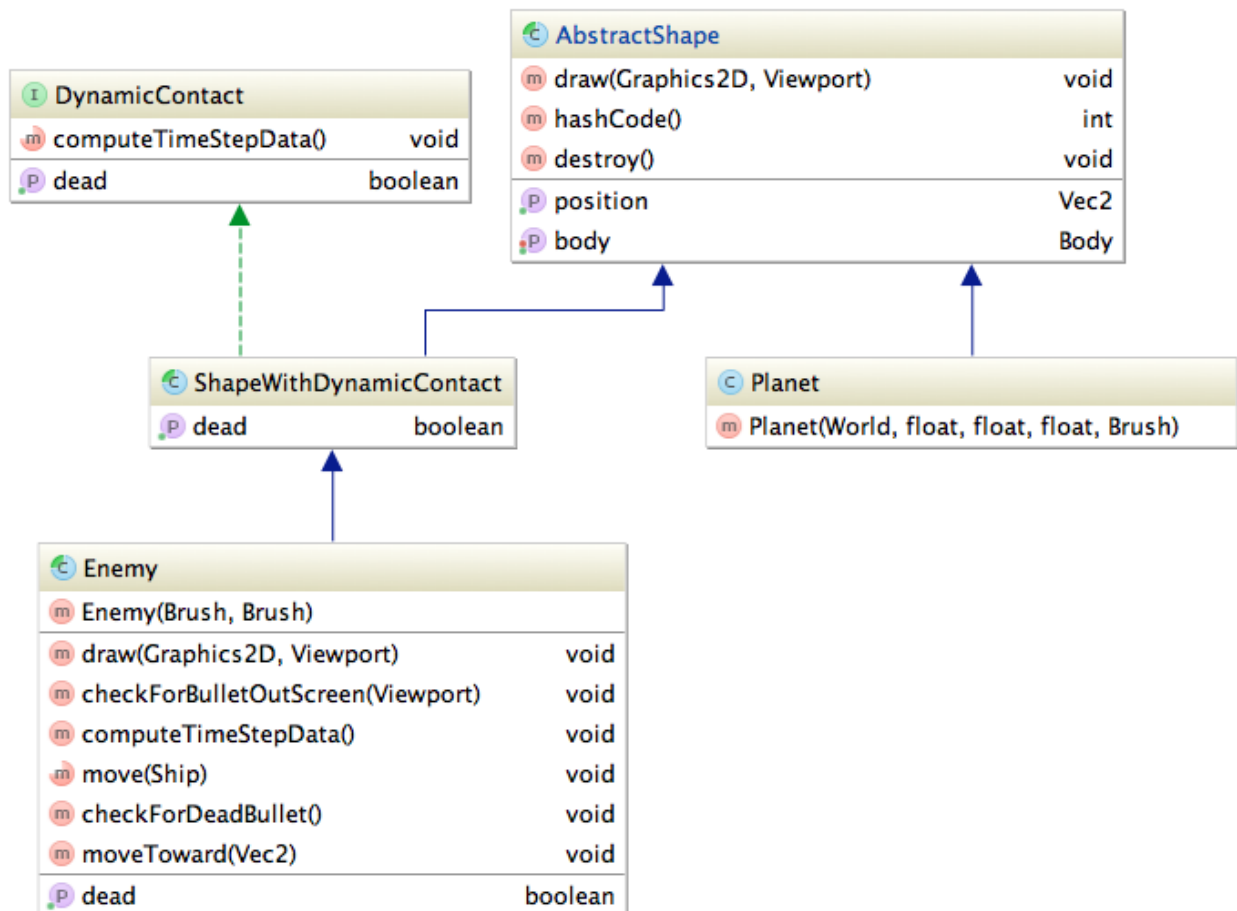
- La mort de tous les ennemis
- La mort du joueur
- La fin du temps imparti

Dans cette boucle on va d'abord gérer les entrées clavier. Par exemple si le joueur appuie sur la touche ↑ du clavier, on va demander au vaisseau d'avancer avec sa méthode `up()`.

Puis on demande au moteur physique de calculer les nouvelles vitesses et les collisions entre les différentes entités du jeu. On traite les résultats des calculs du moteur physique et on affiche toutes les entités qui ont lieu d'être.

L'architecture

Les formes et leur contact



AbstractShape fournit une base pour pouvoir dessiner des éléments du jeu. Ils dérivent tous de cette classe.

DynamicContact représente des éléments JBox2D de type *dynamique* (bodyDef.type = BodyType.DYNAMIC).

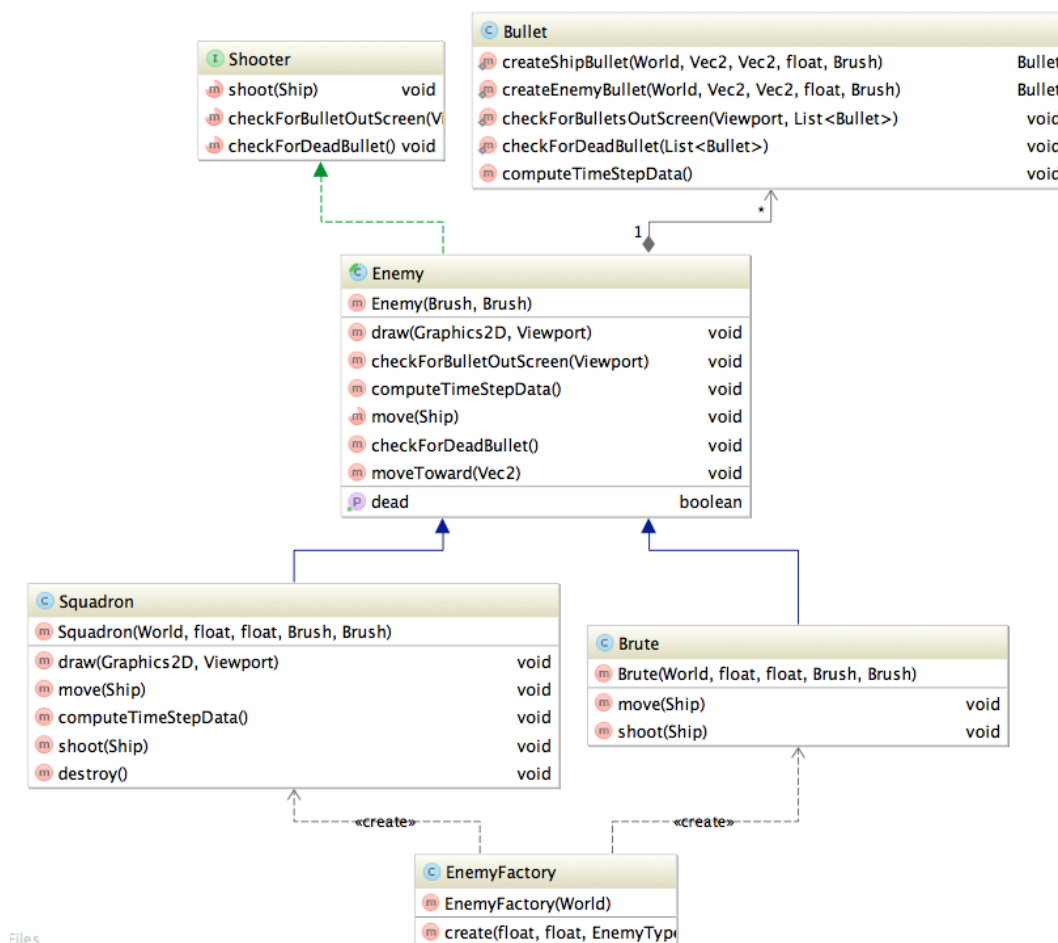
Les types *statiques*, à l'opposé des types *dynamiques* ne sont pas touchés par la gravité ou les forces extérieures. C'est la raison pourquoi Planet n'étend pas DynamicContact alors que Enemy si.

Il s'est avéré que toutes les classes qui étendent AbstractShape implémentent aussi DynamicContact et ont quasiment tous la même implémentation de isDead().

On a donc créé ShapeWithDynamicContact qui étend AbstractShape, implémente DynamicContact, et propose une implémentation par défaut de isDead().

Les classes cités plus haut maintenant étendent directement de ShapeWithDynamicContact.

Ennemis



Ci-dessus on a une vue générale sur les ennemis. Tous étendent Enemy, qui implémente elle-même Shooter.

Les ennemis sont créés à travers la factory EnemyFactory.

Affichage et déroulement du jeu



| C Environment | | |
|---------------|---|-------------|
| m | createEnvironment(World, Viewport, int, int, Configuration) | Environment |
| m | getTimeLeftAsString(long, int) | String |
| m | getTimeLeftAsLong(long, int) | long |
| m | computeDataGame() | void |
| m | noMoreEnemies() | boolean |
| m | draw(ApplicationContext, Viewport, long, int) | void |
| P | ship | Ship |

Environment sert à initialiser le jeu et ses éléments (vaisseau, ennemis, radar, planètes), le mettre à jour quand il le faut (par exemple pour spawner une bombe, envoyer une vague d'ennemis), ainsi que nettoyer les éléments morts (ennemis, bombes, balles).

| C ViewPort | | |
|------------|--|-------|
| m | ViewPort(float, float, float, float) | |
| m | getRelX(float) | float |
| m | getRelY(float) | float |
| m | getClippedCoords(float, float, float, float) | ords |

Un *viewport* est la partie de l'univers affichée à un moment donné.

Cette classe contient donc diverses pour manipuler l'écran, notamment des méthodes de dessin.

| | |
|--|-------------|
|  Game | |
|  Game(Configuration) | |
|  drawGameOver(ApplicationContext, St | |
|  run(ApplicationContext) | void |
|  height | int |
|  width | int |

Game contient principalement la méthode `run()` passée à Zen3. Cette méthode se charge d'initialiser le World de JBox2D, le Viewport, Environment et contient la boucle principale du jeu.

Après la fin du jeu, Game affiche l'écran de fin de jeu pour proposer de quitter ou de rejouer.

Les entités du jeu

Vue d'ensemble

Basiquement toutes les entités du jeu héritent d'une seule et même classe. Elle contient un seul champ de la classe Body qui représente un objet physique de *JBox2D*.

En ce qui concerne le dessin, toutes les entités du jeu se dessinent avec trois méthodes.

En effet un Body contient des Fixture (une courbe de base) et chaque *Fixture* contient sa couleur.

Ainsi lors du dessin il suffit d'explorer toutes les Fixture d'un Body et de les dessiner en fonction de leur type et de leur couleur.

Les ennemis

Comme tous les ennemis ont le même but (détruire le vaisseau du joueur), ils héritent tous d'une même classe abstraite Enemy. Cette classe s'occupe :

- De la gestion des données calculées par le moteur physique
- De la gestion des balles de l'ennemi (suppression, stockage, dessin)
- Et autres...

Exemple : ajouter un nouveau type d'ennemi

Ainsi pour créer un ennemi il suffit d'implémenter sa forme et ses caractéristiques physiques, son déplacement et sa stratégie de tir.

Voici le code pour implémenter un ennemi supplémentaire (disons un carré) :

```
/**
 * Ennemi en forme de carré qui distance le vaisseau
 */
public class Truc extends Enemy {
    public Truc(World world, float x, float y, Brush brush, Brush bulletColor) {
        super(brush, bulletColor); // Appel du super constructeur pour la couleur

        BodyDef bodyDef = new BodyDef(); // On déclare une définition de Body de type
        bodyDef.type = BodyType.DYNAMIC; // DYNAMIC (c'est à dire qu'il est soumis aux lois
        bodyDef.position.set(x,y); // physiques du moteur)

        setBody(world.createBody(bodyDef)); // Appel à la factory du World pour crée un Body
        getBody().setUserData(this);

        // Création d'une Fixture (entité élémentaire qui se rattache a un Body avec une certaine
        // position par rapport a lui)
        FixtureDef fd = new FixtureDef(); // On déclare une définition de Fixture
        fd.density = 1.0f; // Avec une densité de 1
        fd.userData = enemyColor;
        fd.filter.categoryBits = FixtureType.ENEMY; // C'est un ENEMY qu'on crée
        fd.filter.maskBits = FixtureType.BULLET | // Qui peut entre en collision avec BULLET
        FixtureType.SHIP | // ou SHIP
        FixtureType.ARMED_BOMB | // ou ARMED_BOMB
        FixtureType.ARMED_MBOMB | // ou ARMED_MBOMB
        FixtureType.PLANET; // ou PLANET

        PolygonShape polygonShape = new PolygonShape(); // On déclare une forme
        polygonShape.setAsBox(2,2, new Vec2(),0); // On en fait un carré avec un centre en (0, 0)
        fd.shape = polygonShape; // On renseigne la Shape dans la FixtureDef
        getBody().createFixture(fd);
    }

    @Override
    public void move(Ship ship) {
        Vec2 speed = computeFollowingSpeed(ship.getPosition(), // On applique un algo de poursuite
        getBody().getPosition(), // avec un faible coefficient de rappel,
        ship.getLinearVelocity(), // pour suivre de loin
        getBody().getLinearVelocity(),
        0.1f);
        getBody().setLinearVelocity(speed.mul(0.90f)); // On ne prend que 90% de la vitesse obtenue
        // pour pouvoir distancer l'ennemi
    }

    @Override
    public void shoot(Ship ship) {
        // Cet ennemi ne tire pas
    }
}
```


Après ça il suffit juste d'ajouter l'ennemi dans l'enum `EnemyType` et d'ajouter une case au switch dans la méthode `create()` de la classe `EnemyFactory`.

Dans les cinq classes implémentant les ennemis, les premières lignes de code se ressemblent mais ce code n'est pratiquement pas factorisable, ou très peu. De plus il est préférable d'avoir tout le code de construction sous les yeux pour ne rien rater. Le code est un peu long mais les objets ne sont pas si complexes que ça.

Les deux ennemis supplémentaires

- Le triangle qui tourne

Forme : petit triangle.

Déplacement : suit le joueur en conservant une distance minimale.

Stratégie de tir : tire de façon anarchique depuis ses trois angles.

- La brute

Forme : cercle.

Déplacement : fonce sur le joueur, l'envoie valser et s'autodétruit.

Stratégie de tir : aucune, il ne tire pas.

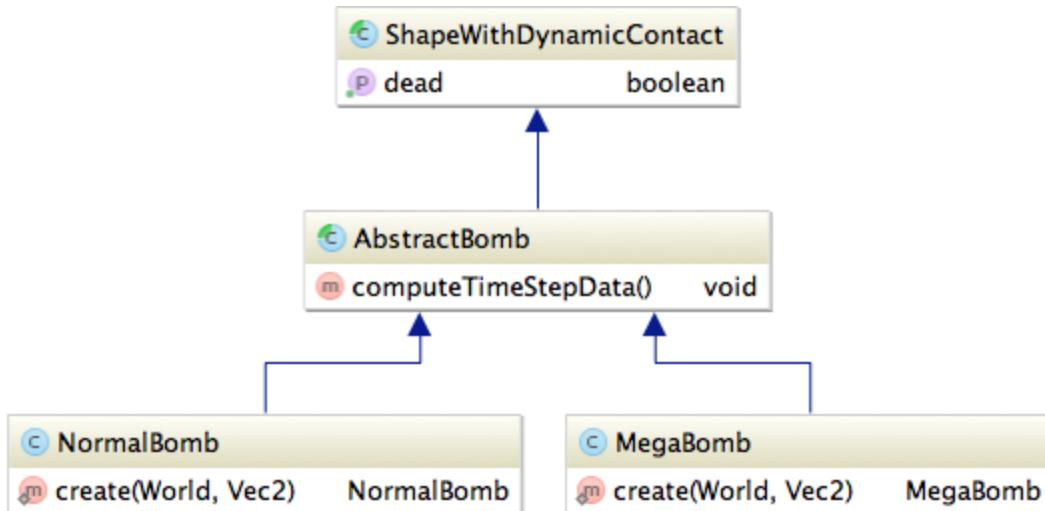
Les Bombes

Il y a deux types de bombes dans le jeu : des bombes dites *normales* qui explosent et tuent les ennemis, et d'autres dites *mega* qui implosent, et donc aspirent les ennemies vers leur centre avant de les avaler.

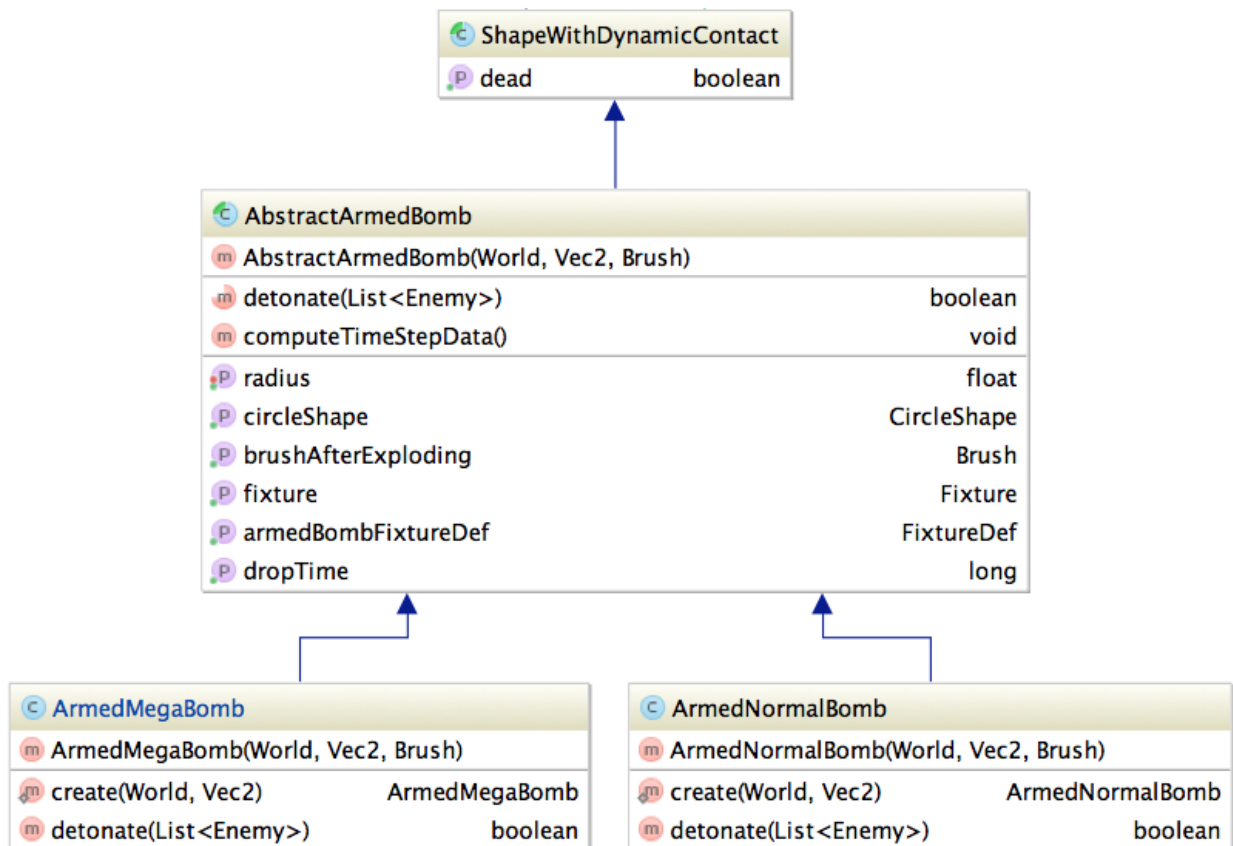
Techniquement, il y a une séparation entre les bombes qui ne sont pas encore prises (*NormalBomb* et *MegaBomb*), et celles qui sont lancées par le vaisseau du joueur (*ArmedNormalBomb* et *ArmedMegaBomb*).

Utiliser une seule classe de bombes pour représenter ces deux types n'a pas de sens car celles-ci ont un fonctionnement fondamentalement différent —simple exemple, l'une peut être récupérée, l'autre pas—, et que faire en sorte que la bombe récupérée par le vaisseau soit la même qu'il stocke, et la même qu'il lance demande une complexité accrue et inutile.

Les bombes qui apparaissent sur la map, sensées être prises par le vaisseau du joueur, ne contiennent pas de logique spécifique. Au contact du vaisseau, sa méthode `addBomb()` ou `addMegaBomb()` est appelée.



Concernant les bombes armées, une fois lancées, leur méthode *detonate()* est appelée à chaque tour de boucle, jusqu'à ce que celle-ci retourne *false* pour signaler que l'explosion est terminée.



Pour le cas de bombe normale, *detonate()* augmente sa taille jusqu'à une certaine valeur pour simuler une explosion. Bien sûr, au contact de l'ennemi, l'onde de choc le détruit.

Concernant la méga bombe, au premier appel de *detonate()*, elle stocke les ennemis les plus proches dans une liste, elle les attire vers elle au fil des appels, avant de les détruire quand ils arrivent à son centre. Quand tous les ennemis sont détruits, ou qu'elle arrive à bout de sa vie (dix secondes), elle disparaît de l'écran.

Les planètes

Les planètes sont générées automatiquement en fonction de la densité décrite dans le fichier de description du niveau. Elles sont ensuite stockées dans une table de hachage, afin de les retrouver si le vaisseau explore à nouveau la même zone de l'espace.

La clé de hachage représente les coordonnées d'une zone, la valeur est une liste de planètes contenue dans cette zone.

De cette façon seul les planètes de la zone courante et des zones alentours sont affichées. Ce système permet de ne pas parcourir une liste, qui pourrait être longue, et de faire des tests sur chaque planète.

Les déplacements

Après avoir demandé (et fait une implémentation de) la possibilité de récupérer les touches lâchées par l'utilisateur dans Zen3, il est devenu possible de faire déplacer le vaisseau du joueur dans n'importe quelle direction, en supportant plusieurs actions simultanées.

La technique utilisée est assez simple : à chaque fois qu'on appuie sur une touche, on la stocke dans un Set, et on la retire une fois lâchée. On exécute ensuite les actions correspondantes à chaque touche dans le Set.

Les collisions

Il y a deux façons de gérer les collisions avec JBox : soit traiter une collection de collisions à chaque tour de boucle, soit utiliser des méthodes de callback en implémentant un Listener.

La solution retenue pour le projet est celle à base de callbacks, en ajoutant les fonctions de filtrage de JBox. C'est-à-dire qu'il est possible de définir que telle entité peut entrer en contact avec telle autre.

Ainsi chaque élément de comportement différent définit un type, chaque élément d'un type spécifique peut définir des collisions avec d'autres types, sans qu'elles soient nécessairement les mêmes que celles d'un autre objet du même type.

Afin de ne pas utiliser des `instanceof` ou faire une longue série de `else if`, une matrice à deux dimensions de lambdas a été créée. Elle dispose de toutes les méthodes afin gérer tous les types de collisions possibles. Ainsi la méthode pour gérer les collisions peut s'écrire de la façon suivante : `action[f1.type][f2.type].accept(f1, f2);`

Le Radar

Comme sur un avion ou un bateau, il est possible de connaître la position des ennemis à proximité sans forcément les voir sur l'écran.

Cette option est très pratique puisque le temps joue contre nous. En effet si on distance de trop un ennemi on ne plus savoir où il est. Si un ennemi est vraiment "trop loin" il peut sortir de l'écran du radar.

Le radar est ni plus ni moins que la représentation de l'écran à une échelle moindre.

Pour implémenter cette fonctionnalité, on a utilisé la classe `OBBViewportTransform`, cette classe permet de transformer les coordonnées de point et de vecteur, pour passer d'une représentation à une autre.

De plus cette transformation est réversible. Cette classe permet de choisir une échelle pour la transformation, ainsi que le décalage du plan et un centrage évolutif (avec `setCenter()`).

Grâce à cette classe, l'implémentation du radar s'est faite assez rapidement et simplement. Il est à noter que c'est aussi cette classe qui permet d'afficher les entités du `World` à l'écran et de recentrer le vaisseau.

La configuration des niveaux

Comme demandé dans le sujet, les niveaux sont configurables avec des fichiers XML.

On a utilisé l'API JAXB pour mapper le fichier de description de niveau à la configuration du jeu dans le programme. Cette API permet de le faire facilement, tout en restant flexible.

Un exemple d'ajout de variable de configuration est disponible plus bas.

La gestion de la configuration est faites en plusieurs parties.

- Déclaration des mappings XML avec des annotations

Exemple :

```
@XmlRootElement
public final class Configuration {
    @XmlElement private int gameDuration;
    @XmlElement private int planetsDensity;
    // ...
    @XmlElementWrapper(name = "enemyWaves")
    @XmlElement(name = "enemyWave")
    private List<EnemyWave> enemyWaves;
    private boolean hardcore;
```

- Définition du schéma XML du fichier de description des niveaux

Cela permet, d'une part, de définir le format des fichiers de niveaux de manière formelle, et d'une autre part de pouvoir valider ces fichiers, pour éviter que l'utilisateur final mette des valeurs non autorisées.

- Écriture du code qui se chargera de *marshaller*¹ le fichier de description du niveau

Il s'agit d'une simple méthode statique `loadFrom(File)` qui transforme un fichier XML pour retourner un objet `Configuration`, validant au passage ce fichier contre le fichier de définition évoqué plus haut (situé dans le dossier *resources* du package *config*).

La partie principale du code est la suivante :

```
Schema schema = sf.newSchema(ConfigurationLoader.class.getResource(
    "resources/level.xsd"));

Unmarshaller unmarshaller = context.createUnmarshaller();
unmarshaller.setSchema(schema);
unmarshaller.setEventHandler((e) -> false); // crash on error
Configuration config = (Configuration) unmarshaller.unmarshal(file);
```

Exemple : ajout d'une nouvelle variable de configuration

Supposons qu'on veuille ajouter une variable de configuration `foo`, qui sera un entier avec une valeur minimale de 0.

Il suffit de la déclarer dans *level.xsd* :

```
<xs:element name="foo">
  <xs:simpleType>
    <xs:restriction base="xs:int">
      <xs:minInclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Puis dans l'ajouter dans la classe *Configuration* :

```
@XmlElement private int foo;
// ...
public int getFoo() {
    return foo;
}
```

¹ On peut utiliser *désérialiser* comme synonyme de *marshalling*

C'est tout. Il est maintenant possible de déclarer `<foo>123</foo>` dans n'importe quel fichier de description de niveau.

Le Synchronizer

Synchronizer est une classe contenant deux méthodes. Elle permet d'attendre un certain laps de temps, afin que la boucle principale dure toujours le même temps.

Ainsi on peut garantir un taux de rafraîchissement d'au plus 60 Hz.

Elle est utilisée dans `Game::run` pour synchroniser la boucle principale du jeu.

Le générateur de vagues d'ennemis

La classe `EnemyWavesGenerator` permet de fabriquer des vagues d'ennemis quand c'est nécessaire et d'indiquer si il n'y a plus d'ennemis à fabriquer.

Comme on peut le voir sur la capture d'écran si dessous, lors de la génération d'une vague, les ennemis sont répartis à une distance égale sur un cercle formé autour du vaisseau. Pour l'exemple la vague a été générée dans l'écran, mais dans le jeu la vague est plus éloignée.

En réalité les ennemis sont écartés d'une *largeur d'écran*² du vaisseau.

² le plus grand entre la largeur et la hauteur

