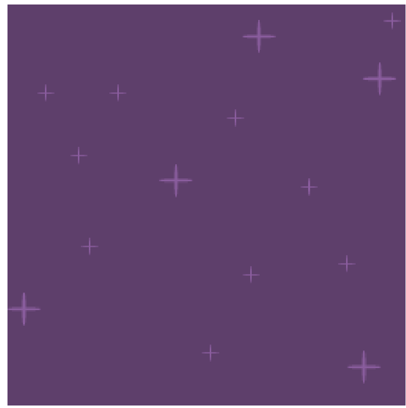
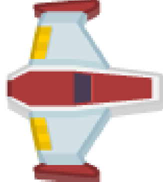
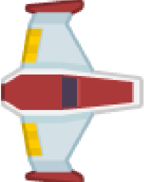
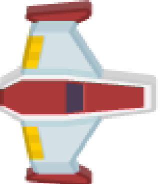
Faculté des Sciences de Montpellier



28 avril 2016
Rapport de projet TER-L3

Jeu de tir avec gravité

Tianome Rasolohery, Mohamed Bareche, Jordan Ferrad

Sommaire

1 Introduction **2**
1.1 Encadrement . . . 2
1.2 Contexte . . . 3

2 Moteur de jeu **4**
2.1 Bibliothèques . . . 4
2.2 Design Pattern . . . 5
2.3 Architecture logicielle . . . 5
2.4 Persistance . . . 6
2.5 Géométrie . . . 6
2.6 Physique . . . 6
2.6.1 Théorie . . . 6
2.6.2 Adaptation . . . 7
2.7 Jouabilité . . . 8
2.8 Objectifs initiaux et problèmes rencontrés . . . 8

3 Interface Graphique **9**
3.1 Cahier des charges . . . 9
3.2 Zones de la fenêtre . . . 9

4 Améliorations prévues **11**
4.1 Optimisations . . . 11
4.2 Jouabilité . . . 11

5 Conclusion **12**

Appendices **13**

A Diagrammes **14**
A.1 Diagramme de cas d'utilisation du joueur . . . 14
A.2 Diagramme de classes . . . 15

B Ressources artistiques **16**
B.1 Audio . . . 16
B.2 Graphique . . . 16

C Usage **17**
C.1 Manuel d'utilisation . . . 17
C.1.1 Gameplay . . . 17
C.1.2 Compilation . . . 19

Chapitre 1

Introduction

1.1 Encadrement

- **Responsable de l'UE** : Christian Retoré.
- **Encadrant** : Jean-Florent Raymond.
- **Attribution du sujet** : Février 2015.
- **Date de retour du présent rapport écrit** : 28 avril 2016.
- **Date de soutenance** : Semaine du 23 mai.
- **Jury de la soutenance** : Christian Retoré, Federico Ulliana.

1.2 Contexte

Dans le cadre de nos études en 3ème année en informatique, nous nous sommes vu proposer une liste de projets afin de parfaire nos connaissances en programmation ainsi que de manière plus générale le développement d'applications.

Nous avons choisis le sujet **"Jeu de tir avec gravité"** car nous sommes intéressés par le traitement de l'image en général et par l'étude de l'architecture d'un moteur de jeu.

Rappel des consignes pour ce projet :

> Le but de ce projet est d'écrire un jeu de tir en 2D où un vaisseau spatial vu de dessus doit se défendre contre des astéroïdes et des ennemis. Un bon exemple est **Kobo Deluxe**.
>
> Une caractéristique requise est la prise en compte de la gravité : chaque objet du jeu (vaisseau, astéroïde, missile, etc.) a une masse et agit sur les trajectoires des autres objets via la force gravitationnelle.
>
> Les participants devront aussi s'intéresser aux graphismes, aux sons et au comportement des ennemis.

Chapitre 2

Moteur de jeu

2.1 Bibliothèques

1. **SFML**
   — la **SFML**[1], écrite par Laurent Gomila[2] est venue en réponse au besoin d'un équivalent de la **SDL**[3] pour des langages orientés objets.
   En effet, cette bibliothèque fut dans un premier temps implémentée pour le C++ uniquement, puis des portages **Java**, **Ruby**, **Python** et bien d'autres langages firent leur apparition.
   Les premiers ont été écrits par L'auteur de la **SFML** et les suivants par des contributeurs.

   Ils sont arrivés à un état stable avec le temps et sont tous portables.

   Cette bibliothèque constitue une surcouche du travail de contexte graphique et de calcul matriciel effectué par l'API **OpenGL**, elle constitue donc une aide à la programmation et non pas une vraie bibliothèque 2D ou 3D comme **Ogre**.

   SFML fait donc continuellement le relais entre les besoins du développeur et les appels aux fonctions d'**OpenGL** en les masquant par des appels a ses propres fonctions et procédures.
   Cependant elle rend possible la création de contexte **OpenGL** manuellement et le rattachement de celui-ci à l'objet fenêtre de **SFML**.

   Enfin, étant destinée à des projets en langages orientés objet, elle est-elle même implémentée de cette manière et rend facile la création de nouvelles entités personnalisées en héritant d'une ou de plusieurs de ses classes.

1. Simple and Fast Mutlimedia Library
2. Membre du site www.Develloppez.net
3. Simple DirectMedia Layer

2.2 Design Pattern

Le Design pattern[4] utilisé dans ce projet est une fusion entre le **pattern Singleton**[5] et le **pattern MVC**[6]
Effectivement, la fonction principale **main()** instancie un `Jeu` qui dans son constructeur, remplis sa collection hétérogène de `Phase` en les créant dynamiquement et en les ajoutant a son `vector<Phase>` de la manière suivante :

```
Dans le fichier header :
        vector<Phase*> listePhases;

Dans le constructeur
        listePhases.push_back(new Intro(...,...,...));
        listePhases.push_back(new Menu(...,...,...));
        listePhases.push_back(new Options(...,...,...));
        listePhases.push_back(new Partie(...,...,...));
        listePhases.push_back(new Pause(...,...,...));
        ...
```

Les phases créées dynamiquement sont des **singletons** car une phase a besoin d'être instanciée pour avoir son existence propre, mais pas plus d'une fois, ce qui explique l'utilisation des **singletons**.
L'utilisation du model **MVC** quant à lui est expliqué par le fait que le `Jeu` fait office de contrôleur et propage les changements a tout le programme car, à la création chaque phase reçoit en paramètre un pointeur vers le jeu, pour pouvoir accéder à ses membres.

2.3 Architecture logicielle

Les objets de jeu se déterminent avec la logique suivante :

La classe `Objet` contient les calculs physiques et dérive vers les classes
- `Destructible`
- `Collectable`

qui constituent deux implémentations différentes des actions à réaliser en cas de collision, `Destructible` introduis la notion de durée de vie avec une quantité de points de vie pour l'`Objet`.

`Destructible` dérive à son tour vers :

- `Joueur` qui permet la navigation de l 'entité via les périphériques clavier,souris,`joystick`.
- `Ennemi` qui rend possible un comportement de pseudo IA[7].

4. arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel
5. *voir la source* Devoloppez.net
6. Model Vue Contrôleur
7. Intelligence Artificielle

Cette structure permet d'avoir une collection hétérogène de `Destructible` qui sont tous de même **type statique** mais de **type dynamique** dérivés, et ainsi, de pour voir la parcourir en profitant du comportement associé a la **liaison dynamique**.

2.4 Persistance

Le démarrage de l'application avec les bons paramètres ainsi que la mémorisation de ces derniers se fait via la sauvegarde dans un simple fichier texte, grâce à une classe définie dans le fichier **gestion.hpp**.

Elle utilise les flux **ifstream** et **ofstream** pour réaliser le *parsing* en lecture et en écriture. La structure du fichier de sauvegarde de configuration est simple, il s'agit d'une suite de lignes qui contiennent chacune un couple clé-valeur séparés par un séparateur de type **string**.

L'algorithme de parcours est donc aussi simple car il consiste à parcourir les lignes jusqu'à la rencontre du bon paramètre.Il faut alors remplacer la valeur adjacente à ce paramètre par la nouvelle. Cette sauvegarde se fait par la réécriture du fichier dans un fichier temporaire qui sera ensuite renommé par appel système.

2.5 Géométrie

L'ensemble des algorithmes utilisés pour résoudre les problèmes liés à la trigonométrie seront explicités lors de la présentation du projet.

2.6 Physique

2.6.1 Théorie

Calcul général

Selon le Principe fondamental de la dynamique, à un instant T, la somme des forces appliquées à un corps est égale à sa masse multipliée par son accélération :

$$\boxed{\vec{a} = \frac{1}{m} \sum \vec{\mathrm{F}}_i}$$

On a pu déduire de cette formule le **vecteur accélération** de l'objet, puis en intégrant celle-ci, nous avons pu calculer le **vecteur vitesse**, puis le **vecteur position**.

Collisions

Les collisions entre les objets de notre application obéissent au principe du **Choc élastique** qui consiste en la modification des trajectoires de chacun des deux corps avec un échange total et sans perte de leur énergie cinétique et sans modification de leur masse.

$$\boxed{\vec{w}_1 = \|(\frac{m_2}{m_1+m_2}).(\vec{v}_1 - \vec{v}_2)\|.\widehat{v_2}}$$

$$\boxed{\vec{w}_2 = \|(-\frac{m_1}{m_1+m_2}).(\vec{v}_1 - \vec{v}_2)\|.\widehat{v_1}}$$

Attraction gravitationnelle

Le calcul de l'attraction gravitationnelle entre les objets s'effectue selon la **Loi universelle de la gravitation** :

$$\boxed{F_{A/B} = F_{B/A} = G\frac{M_A M_B}{d^2}N}$$

Où **G** est la constante universelle de la gravitation et **N** est l'unité de mesure, le **Newton**.

2.6.2 Adaptation

L'algorithme général sur la physique est implémenté au sein de la classe *Objet* et se décompose en plusieurs méthodes séparées qui sont appelées lors du parcours itératif de la liste des objets :

1. *initialiser().*
   - Affectation du vecteur nul au vecteur de la somme des forces.
   - Ré initialisation des variables booléennes qui servent à l'optimisation du parcours des objets.
2. *attraction().*
   - implémentation de la formule de Newton de la Loi universelle de la gravitation.
   - optimisation de cette méthode via l'implémentation du calcul du barycentre de tous les objets distants.
3. *collision().*
   - implémentation de la formule du choc élastique
4. *debordement().*
   - méthode qui empêche le dépassement de l'écran.
5. *appliquer()*
   - C'est ici que le PFD[8] nous permet de calculer le vecteur accélération de l'objet en divisant la somme des forces accumulées lors des méthodes ci-dessus par la masse de l'objet.
   - Le prochain vecteur vitesse est ensuite calculé en intégrant le vecteur accélération.
   - La prochaine position de l'objet est obtenue en intégrant le vecteur vitesse.

8. Principe fondamental de la dynamique

2.7 Jouabilité

Atténuation de la vitesse

Puisque le vaisseau spatial pouvait atteindre une vitesse illimitée si le joueur appuyait continuellement sur les touches de déplacement (voir en annexe le manuel d'utilisation), nous avons décidé de limiter cette vitesse et de l'atténuer en la multipliant par 0.9999 à chaque instant pour simuler une force de frottement afin d'améliorer la jouabilité.

Restriction de la zone d'application de la gravité

Initialement, la force gravitationnelle est calculée à chaque séquence pour chaque objet et ce, quel que soit la distance entre eux.
Il a donc fallu restreindre cette force à une zone délimitée par une distance autour de l'objet.
En effet si un objet est lourd et très éloigné du vaisseau il ne doit pas modifier la trajectoire de celui-ci.

2.8 Objectifs initiaux et problèmes rencontrés

Calcul général

Au départ, nous avions tenté d'effectuer l'ensemble des calculs physiques cités ci-dessus pour chaque objet l'un après l'autre ce qui n'étais pas correct sur le plan physique car la modification des attributs d'un objets entrainait un calcul différent pour les objets suivants.
Nous avons donc rectifié notre algorithme en effectuant chaque partie du calcul séparément pour chaque objet.

Collision

Premièrement, nous avons réalisé les collisions entre deux objets en ajoutant une force a chacun d'entre eux en vue de modifier leur trajectoire lors de la répercussion de la somme des forces.

Par la suite, nous avons préféré opter pour une modification directe des vitesses après collision, en fonction des vitesses avant collision, en utilisant la formule du choc élastique.

Chapitre 3

Interface Graphique

3.1 Cahier des charges

- L'interface graphique regroupe les éléments qui n'ont pas d'interaction physique avec les objets du jeu (Vaisseau, astéroïdes, ennemi, etc.). Les collisions avec les objets graphiques de l'interface n'influent en rien le cours du jeu lui-même.

- Elle donne au joueur la possibilité de se tenir informé par rapport à l'état de son vaisseau (les dégâts qu'il a subi au cours de la partie) et le nombre de points de vie qu'il lui reste .

  C'est-à-dire que si le nombre de points de vie est égal à zéro le joueur a perdu et la partie se termine mais s'il a gagné des objets (**Collectables**) lui offrant des points de vie, ce nombre s'augmentera automatiquement.

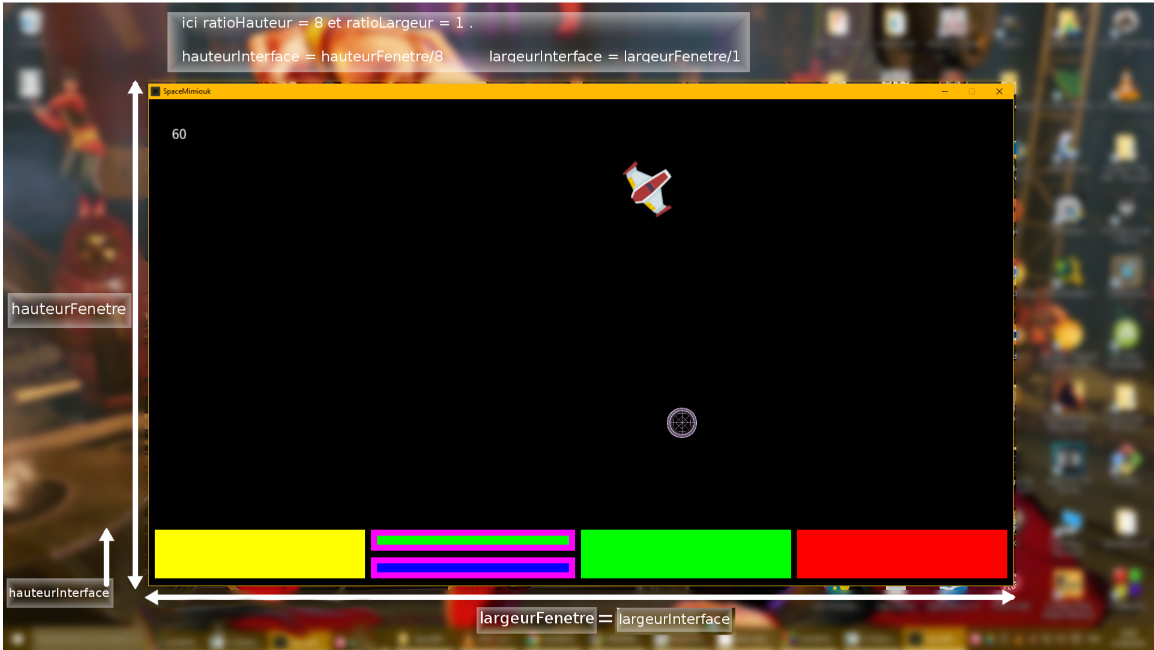
  De même, pour les armes et les munitions, le joueur pourra connaitre les différentes armes qu'il possède (lasers, missiles, etc.) et le nombre de leurs munitions via l'interface graphique.

3.2 Zones de la fenêtre

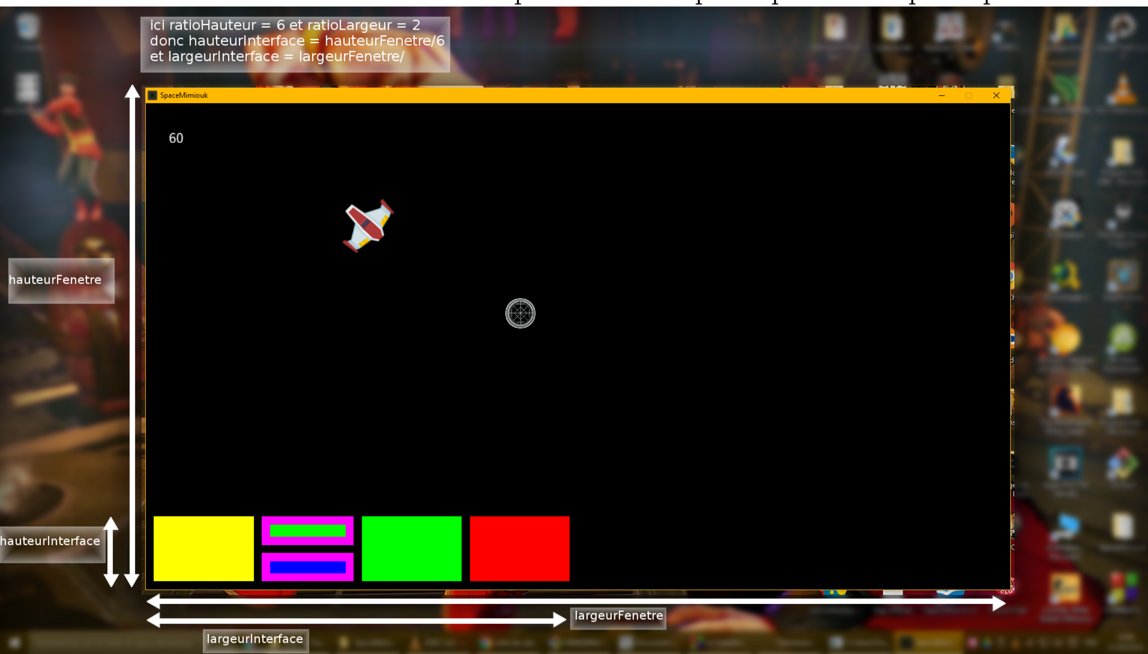
L'écran est séparé en deux zones.
Le joueur évolue dans la **zone principale** sur laquelle se superpose la **zone de l'interface graphique** dans la partie basse de l'écran et qui contient tous les autres objets graphiques permettant l'affichage des caractéristiques du joueur à chaque instant de la partie.

Afin d'appliquer la même procédure à chaque changement de mode d'écran pour repositionner tous les éléments, on a pris en compte la **hauteur de la fenêtre** et la **largeur de la fenêtre** que l'on a divisé par des **coefficients diviseurs** pour avoir une zone qui sera dépendante de la taille de la fenêtre et qui automatiquement va s'ajuster quel que soit le mode de fenêtrage de l 'application (**Mode fenetré** ou **Mode plein écran**).

Cette première image montre un aperçu de cette interface graphique avec 1 comme **coefficient diviseur de largeur** (sa largeur est donc celle de la fenêtre divisée par 1) et le **coefficient diviseur de hauteur** qui a pour valeur 8 (sa hauteur est donc d'un huitième de celle de la fenêtre).



Sur cette seconde image on obtient une interface graphique différente simplement en donnant au **coefficient diviseur de largeur** la valeur **2** pour que la largeur de l'interface soit la moitié de celle la fenêtre.
Le **coefficient diviseur de hauteur** à pour valeur **6** pour qu'elle soit plus épaisse.



Finalement,on a pu établir un mécanisme de positionnement de toutes les images grâce à l'utilisation d'une **marge interne** calculée en fonction de la hauteur de l'interface divisée par un **coefficient diviseur de marge interne** qui sert à positionner tous les composants les uns par rapport aux autres.

Chapitre 4

Améliorations prévues

4.1 Optimisations

Plusieurs optimisations possibles du programme nous sont déjà venu à l'esprit au fur et à mesure de l'implémentation, parmi les suivantes :

- Passage vers un design Pattern plus souple où chaque module(phases,jeu) contiendra un pointeur vers chacun des autres modules et pourra demander directement à celles-ci d'effectuer une méthode.Le jeu ne servira donc plus d'intermédiaire à la propagation de chaque événement modifiant l'état de toute l'application.
- Utilisation de plus de collections pour permettre une plus grande abstraction du code source et une plus grande automatisation des procédures, et donc la factorisation du code.
- Utilisation de collections plus diverses et adaptées.
- implémentation d'un système de particules.
- implémentation d'un système de lecture de vidéos via la bibliothèque de lecture de flux **libvlc**, pour améliorer l'introduction.

4.2 Jouabilité

Concernant les principes de jeu les amélions à venir sont cernées :

- Augmenter la durée du niveau.
- Pouvoir jouer à plusieurs (insertion d'un deuxième joueur)
- Ajouter différentes armes pour enrichir l'expérience.
- Étendre la partie à plusieurs niveaux.
- Améliorer le comportement des ennemis, créer de nombreux ennemis différents.
- Ajouter une couche réseau.
- ...

Chapitre 5

Conclusion

Ce projet nous a conforté dans notre goût de la programmation à but audiovisuel même si nous avons rencontré des épreuves et des choix difficiles tout le long de sa réalisation.
En effet, Le choix de l'architecture a été l'objet de nombreuses et longues discussions car il fut déterminant pour le bon déroulement du projet.

Au final, nous avons réussi à utiliser un type d'architecture de moteur de jeu qui est peu utilisé par les grandes firmes et qui est celui qui fut utilisé par la majorité dans anciens jeu vidéo des années 80 à 2000, et qui reste très efficace et largement suffisant pour un moteur 2D.

Nous avons implémenté un moteur de jeu et non pas uniquement un jeu dans le sens ou une grande partie des classes écrites durant le projet ont été volontairement écrites avec un souci de l'abstraction, du paramétrage et de la réutilisation.

Ce projet nous a également permis de découvrir comment implémenter des formules physiques au sein d'un programme en les discrétisant.
Ce projet fût vraiment très enrichissant et nous allons le poursuivre et l'améliorer.

Nous adressons nos remerciements à notre encadrant Jean Florent Raymond pour son aide au développement et pour nous avoir aiguillé dans nos priorités.

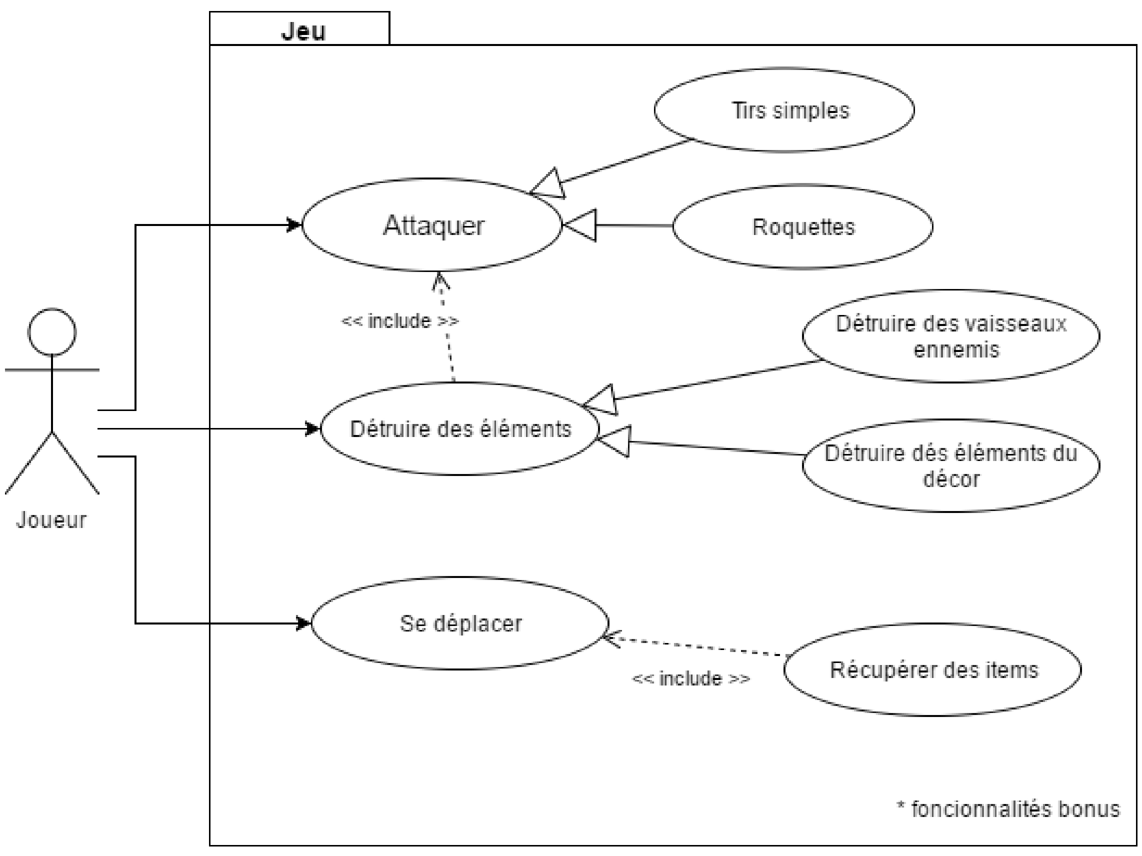
Appendices

Annexe A

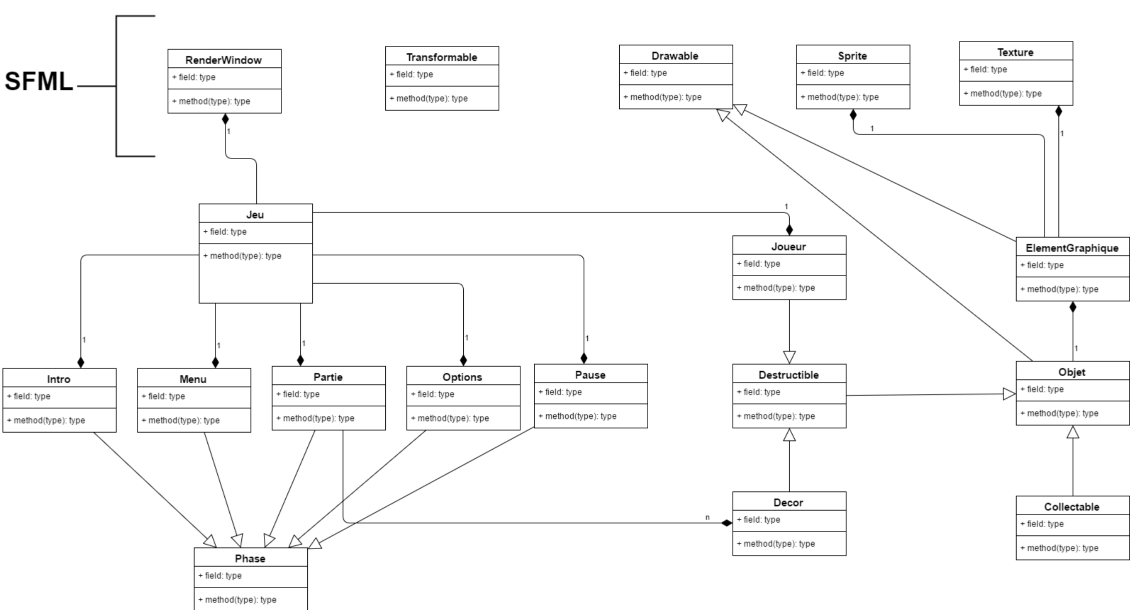
Diagrammes

Voici quelques diagrammes qui résument les explications précédentes.

A.1 Diagramme de cas d'utilisation du joueur



A.2 Diagramme de classes



Annexe B

Ressources artistiques

B.1 Audio

Nous avons composé un petit répertoire musical de quelques musiques, cet aspect est souvent mis de côté dans la production de jeux vidéo et reste néanmoins un atout considérable.

B.2 Graphique

Les éléments d'interface sont entièrement créés sous GIMP, à savoir :
- les boutons
- le curseur
- les jauges paramétrables

Tandis que pour les Sprites [1], ils sont en partie issus du site **opengameart.org** et sont libres de droits.

1. Nom donné aux images qui constituent les éléments d'un jeu

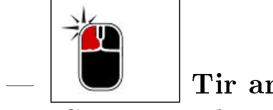
Annexe C

Usage

C.1 Manuel d'utilisation

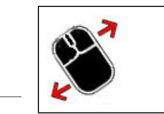
C.1.1 Gameplay

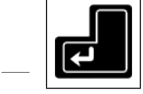
Voici le descriptif des actions liées au différents entrées clavier, souris, et joystick :

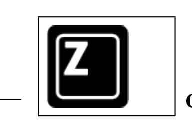
-  **Tir arme spéciale**.
  Consomme des munitions.

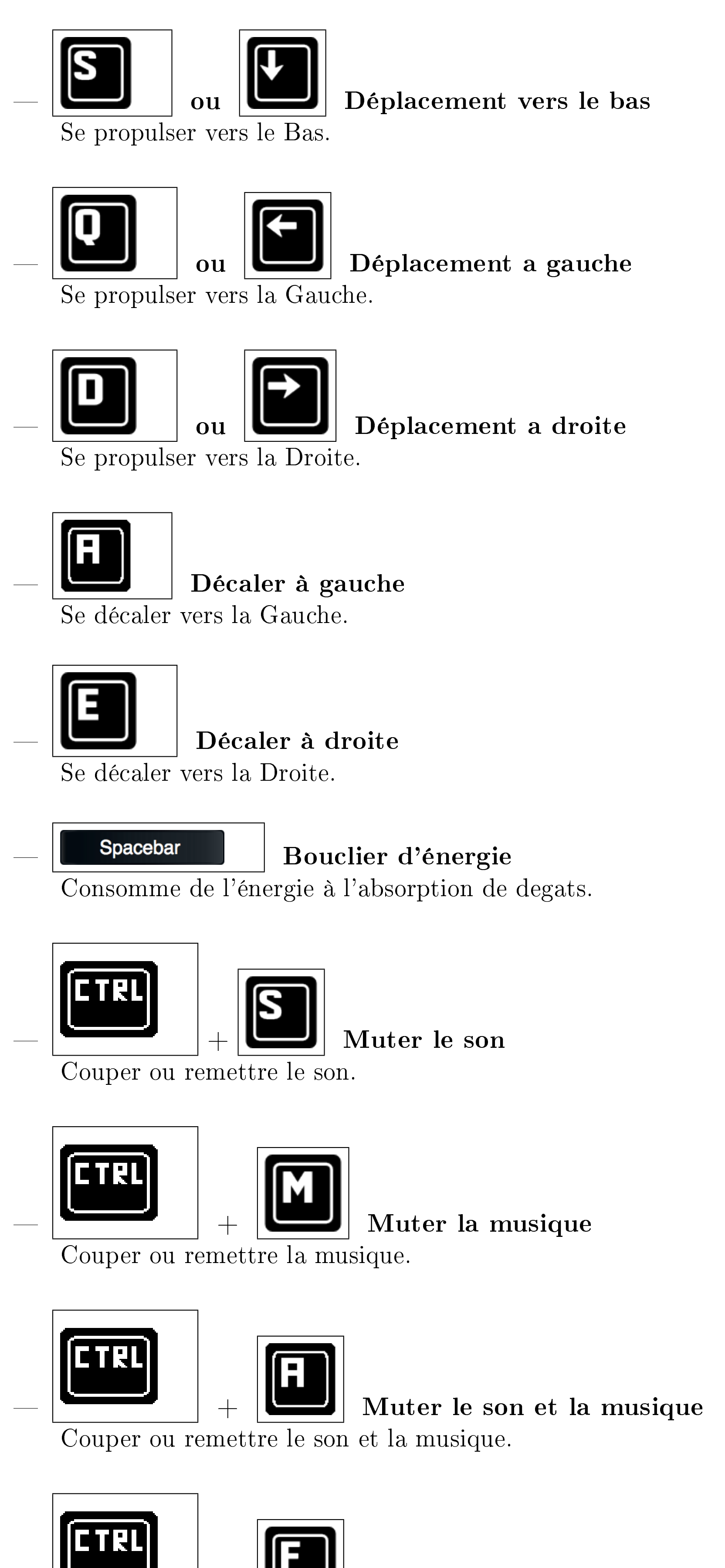
- **Tir laser**
  Consomme peu d'énergie, (appui continu possible).

- **Zoomer-dézoomer**.
  Se rapprocher ou reculer du vaisseau.

-  **Viser**.
  Viser avec le curseur de tir.

-  **Pause**.
  Endormir l'application et ne plus consommer de processeur.

-  **ou** **Déplacement vers le haut**
  Se propulser vers le Haut.



— S ou ↓ **Déplacement vers le bas**
Se propulser vers le Bas.

— Q ou ← **Déplacement a gauche**
Se propulser vers la Gauche.

— D ou → **Déplacement a droite**
Se propulser vers la Droite.

— A **Décaler à gauche**
Se décaler vers la Gauche.

— E **Décaler à droite**
Se décaler vers la Droite.

— Spacebar **Bouclier d'énergie**
Consomme de l'énergie à l'absorption de degats.

— CTRL + S **Muter le son**
Couper ou remettre le son.

— CTRL + M **Muter la musique**
Couper ou remettre la musique.

— CTRL + A **Muter le son et la musique**
Couper ou remettre le son et la musique.

— CTRL + F **Afficher le framerate**
Cacher ou afficher l'information de framerate à l 'écran.

C.1.2 Compilation

Voici les instructions pour les différentes compilations et exécution retrouvables dans le fichier README.txt :

```
COMPILATION LINUX :
        sudo apt-get install libsfml-dev
        cd SpaceMimiouk/
        make sfml
        ./bin/SpaceMimiouk

COMPILATION WINDOWS (codeblocks) :
        ouvrir SpaceMimiouk.cbp
        compiler et executer(F9)

ALL :
        régénérer tous les fichiers «.o» après chaque changement de plateforme:
                make cleansfml (linux)
                rebuild avec codeblocks(windows)
```