Projet L1 – C.M.I.

Spécialité Informatique

Space Invader

Par : BUNEL Paul
FAURE Thomas

Soutenu le : 19/04/2019

Sommaire

I- Introduction

Dans le cadre du module HLSE205 – Projet CMI du second semestre en L1 CMI Informatique, nous avons dû réaliser un jeu vidéo, l'objectif de fin de semestre étant de rendre un jeu fonctionnel, ainsi qu'une vidéo de présentation du projet et un rapport final. Notre groupe était un duo, constitué de BUNEL Paul et de FAURE Thomas, et nous avons choisi de coder une réplique d'un Space Invader.

Le codage du jeu s'est déroulé du 4 février au 12 avril 2019, avec 19,5 heures en TP mais également (et surtout) beaucoup de travail personnel en hors présentiel.

Pour la réalisation de ce jeu, nous avons eu besoin de faire beaucoup de recherches, notamment sur la gestion d'une interface graphique et la création de classes. Nous devions également faire preuve de travail en collaboration, ainsi qu'apprendre à gérer un projet en groupe.

A) Le sujet

L'idée du sujet nous est venue assez rapidement. En effet, nous cherchions quelque chose d'un peu plus ambitieux que le projet fait en ISN en Terminale S (un jeu de morpion pour Paul BUNEL) mais en même temps accessible pour nous, débutants en programmation.

Nous sommes donc partis pour coder un jeu de Space Invaders, un jeu dans lequel le joueur contrôle un vaisseau spatial situé en bas de l'écran, et doit tirer sur des aliens qui descendent progressivement du haut de l'écran.

Le joueur peut uniquement se déplacer de droite à gauche et inverserment, et tirer des projectiles. Il gagne un certain nombre de point à chaque alien tué, l'objectif étant d'accumuler un maximum de points avant de mourir. On ne peut donc pas « gagner une partie » à proprement parler, le joueur peut seulement survivre le plus longtemps possible et effectuer le meilleur score possible.

Le joueur perd s'il est touché par 3 projectiles aliens, ou bien si un alien atteint le bas de l'écran.

B) Cahier des charges

Le jeu doit être sous forme d'interface graphique, et non sur la console.

Au démarrage du jeu, un écran d'accueil donne le choix au joueur de lancer une partie, obtenir des informations sur les règles du jeu, ou quitter l'application.

En jeu, l'utilisateur contrôle avec le clavier un vaisseau spatial, situé au bas de l'écran, qui peut se déplacer horizontalement et envoyer des projectiles.

Un groupe de 55 aliens (11 x 5) doit effectuer un mouvement bien précis : le groupe se déplace de manière uniforme vers un des côtés latéraux de l'écran ; une fois l'extrémité atteinte, le

groupe descend vers le bas sur une longueur égale à la hauteur d'un alien puis se déplace directement vers le côté latéral opposé. Ce schéma se répète en boucle tant qu'il y a des aliens à l'écran. De plus, les aliens peuvent tirer de manière aléatoire (il ne peut y avoir qu'un seul tir à la fois), et ils déplacent de plus en plus vite.

Le projectile tiré par le joueur doit pouvoir éliminer les aliens, un par un. Chaque alien tué rapporte un certain nombre de point au joueur. Le joueur ne peut pas tirer si son projectile est visible sur l'écran.

Les projectiles tirés par les aliens doivent pouvoir enlever une vie au joueur. Le joueur possède 3 vies au début de la partie, celle-ci se termine quand le joueur atteint 0 vie ou quand les aliens atteignent le bas de l'écran.

Si le joueur détruit tous les ennemis, un nouveau groupe de 55 aliens apparaît, et ce jusqu'à ce que le joueur perde.

En parallèle aux fonctionnalités précédentes, une musique de fond doit être jouée aussi longtemps que l'application est ouverte. De plus, des bruitages doivent être joués lors d'un tir (du joueur ou d'un ennemi), ainsi que lors de l'élimination du joueur ou d'un alien.

II- Organisation du projet

A) Organisation du travail

Lors de la première séance, nous avons réfléchi à une manière de découper le projet et de se répartir les tâches. Cependant, mon camarade Thomas Faure a peu à peu décroché de la formation, et je (Paul Bunel) me suis donc occupé de l'intégralité du code du projet, ainsi que de la réalisation du rapport et final et de la vidéo de présentation.
Thomas a toutefois réalisé les différentes textures utilisées dans le jeu, et s'est occupé de rechercher des bruitages et musiques libres de droits.

J'ai donc finalement réparti la tâche qui m'attendait en plusieurs sous parties, que j'ai effectué les unes à la suite des autres, que ce soit en séance de TP ou en travail hors séances.

Voici donc comment s'est organisé mon travail :

1) Prémices : Installation de Code::Blocks et de SFML, recherches sur l'utilisation de SFML
2) Premiers prototypes : création d'un squelette de jeu avec des formes rectangles, un seul alien et un programme en programmation impérative
3) Création des classes : une classe Vaisseau, une classe Projectile et une classe Invader, représentant respectivement le joueur, un projectile et un alien. Ces classes devront gérer les déplacements, collision des différents objets ainsi que certaines variables comme les spirtes, positions, etc.
4) Fonction du schéma de déplacement automatique et de tir aléatoire pour les aliens
5) Gestion et affichage du score et des vies
6) Menu principal et partie audio (bruitages et musiques).

B) Outils de développement

Pour coder ce jeu, j'ai choisi d'utiliser le langage C++, en lui associant la bibliothèque graphique SFML 2.5.1. J'ai fait ce choix car c'est un langage que nous avions vu en cours, qui est très utilisé dans le monde professionnel, c'était donc tout à mon avantage de m'entrainer sur ce langage en faisant ce projet. De plus, j'avais envie de changer du python, qui était le seul langage que j'avais appris jusqu'ici.

Pour la bibliothèque graphique, je n'en connaissais aucune, j'ai donc choisi de me renseigner sur la SFML d'après les conseils de mon professeur référent. C'est donc cette bibliothèque que j'ai utilisé pour mon projet, après de nombreuses recherches sur internet.

Ensuite concernant les logiciels, je me suis servi de Code::Blocks pour gérer le projet, en utilisant le compilateur de base (GNU GCC Compiler) car c'est un des logiciels les plus répandu, pratique pour gérer des projets avec plusieurs fichiers et l'installation de la SFML était relativement peu compliquée.

Nous nous sommes également servi de Google Drive pour se transmettre des fichiers, principalement pour les textures de Thomas.

III-Analyse du projet

A) Analyse préalable

Tout d'abord, j'ai décidé d'emblée de faire mon programme sous forme d'interface graphique et en programmation orientée objet, deux domaines totalement nouveaux pour moi. J'ai choisi la programmation orientée objet car cela me faciliterait l'organisation et la compréhension du code, notamment pour le « rangement » des fonctions, mais également car j'avais envie d'apprendre à me servir des classes et objets.
La quasi-totalité du projet a donc été consacrée à ces deux aspects du code, et donc aux recherches sur l'utilisation de SFML et des classes en C++.

Le programme est constitué de trois classes principales :

- une classe Vaisseau qui contient les paramètres et fonctions du joueur
- une classe Projectile qui contient les paramètres et fonctions du projectile, cette classe sera uniquement instanciée en attribut des 2 autres classes
- une classe Invader qui contient les paramètres et fonctions d'un alien. Cette classe servira pour créer un tableau d'Invader qui représentera le bloc d'alien à détruire.

Chacune de ces classes fait la gestion de plusieurs variables, notamment les sprites/RectangleShape qui sont affichés à l'écran. De plus, elles sont toutes imbriquées les unes dans les autres, ce qui, par ailleurs, a causé de nombreuses difficulté lors de leur programmation.

Le programme et les classes doivent également gérer plusieurs fonctions clés, comme les déplacements et les tirs, gérées par les classes, ou les systèmes de score, vie, affichage d'informations à l'écran, etc., gérés par le programme principal.

Le code a donc été réparti en 8 fichiers différents : 4 fichiers sources : main.cpp (le programme principal), Vaisseau.cpp, Projectile.cpp et Invader.cpp (chacun contenant la définition des méthodes de sa classe) ; et 4 fichiers header : fonctions.h (contenant les fonctions utilisées dans le programme principal), Vaisseau.h, Projectile.h et Invader.h (chacun contenant les attributs et prototypes de sa classe).

B) Analyse détaillée

a) Gestion des différents objets

Les 3 principaux types objets en jeu sont : le joueur, le(s) projectile(s) et les aliens. Dans notre code, chacun de ces objets est créé à partir d'une classe. L'affichage de ces objets à l'écran se fait dans la fonction « display » du fichier main : celle-ci récupère les attributs sprite des différents objets à travers une méthode accesseur.

La classe vaisseau est constituée des méthodes et attributs suivants :

```
class Vaisseau
{
    /*Déclaration de la classe Vaisseau qui nous servira à créer notre joueur.
    Ce qui est affiché à l'écran est son attribut "m_rect", qui est un sprite.
        Elle a également un objet de la classe Projectile en attribut.*/
public: //Méthodes
    inline Vaisseau();
    inline void create(const sf::Texture& t_player);
    inline sf::Sprite getRect() const;
    inline sf::RectangleShape getProjectile() const;
    inline int getScore() const;
    inline int getLives() const;
    inline int getWidth() const;
    inline int getHeight() const;
    inline void resetKill();
    inline void setLives(int vie);
    inline void setPlayerPosition(sf::Vector2f vect);
    inline void resetProj();
    inline void tirer(Invader tabInv[11][5], int tailleI,
        int tailleJ, bool *tir, sf::Sound& s_explosion, float *sptr);
    inline void deplacement_vaisseau();
    inline void score(int s);
    inline void live();

private: //Attributs
    int m_points;
    int m_vies;
    sf::Vector2f m_positionVaisseau;
    sf::Sprite m_rect;
    int m_height;
    int m_width;
    Projectile m_projectile;
};
```

Déclaration de la classe Vaisseau (fichier Vaisseau.h)

Si on enlève les accesseurs et les mutateurs, la classe n'est en fait constituée que de 2 méthodes principales (hormis le constructeur) : « tirer » et « deplacement_vaisseau ». Ces méthodes au nom intuitif sont appelées dans une fonction « jouer » du programme principal. La fonction « tirer » comporte beaucoup de paramètre, dont la plupart servent à appeler la méthode « fcollision » de l'attribut m_projectile du vaisseau.

Ensuite vient la classe Invader, constituée des méthodes et attributs suivants :

```
class Invader
{
    /*Déclaration de la classe Invader, qui représentera les aliens (les ennemis).
    Ce qui est affiché à l'écran est son attribut "m_inv", qui est un sprite.
        Elle a également un objet de la classe Projectile en attribut*/
public: //Méthodes
    Invader();
    sf::Sprite getInv() const;
    int getWidth() const;
    int getHeight() const;
    sf::RectangleShape getProjectile() const;
    void setProjPosition(sf::Vector2f vect);
    void setInvPosition(sf::Vector2f);
    void moveInv(float x, float y);
    void create(const sf::Texture& t_invader1a);
    void resetProj();
    void tirer(bool *tir_alien, Vaisseau *player, sf::Sound& s_mort);

private: //Attributs
    sf::Sprite m_inv;
    int m_width;
    int m_height;
    Projectile m_projectile;
    sf::Vector2f m_position;
};
```

Déclaration de la classe Invader (fichier Invader.h)

Comme pour la classe Vaisseau, si on retire les accesseurs et mutateurs, il nous reste les méthodes « moveInv » et « tirer ». Ces deux méthodes seront gérées de manière particulière par le programme principal, la première car il faut déplacer un bloc entier d'alien en même temps, mais aussi car ils changent de direction à chaque fois qu'ils atteignent un bord de l'écran ; et la 2e car il faudra gérer un système pour faire tirer un alien aléatoire au bout d'un certain temps lui aussi aléatoire. La première méthode sera donc utilisée dans la fonction « deplacementInvader » (en association avec la fonction « sensDeplacement ») tandis que la 2e sera utilisée dans la fonction « jouer ».

Enfin, la dernière classe est la classe Projectile, constituée des méthodes et attributs suivants :

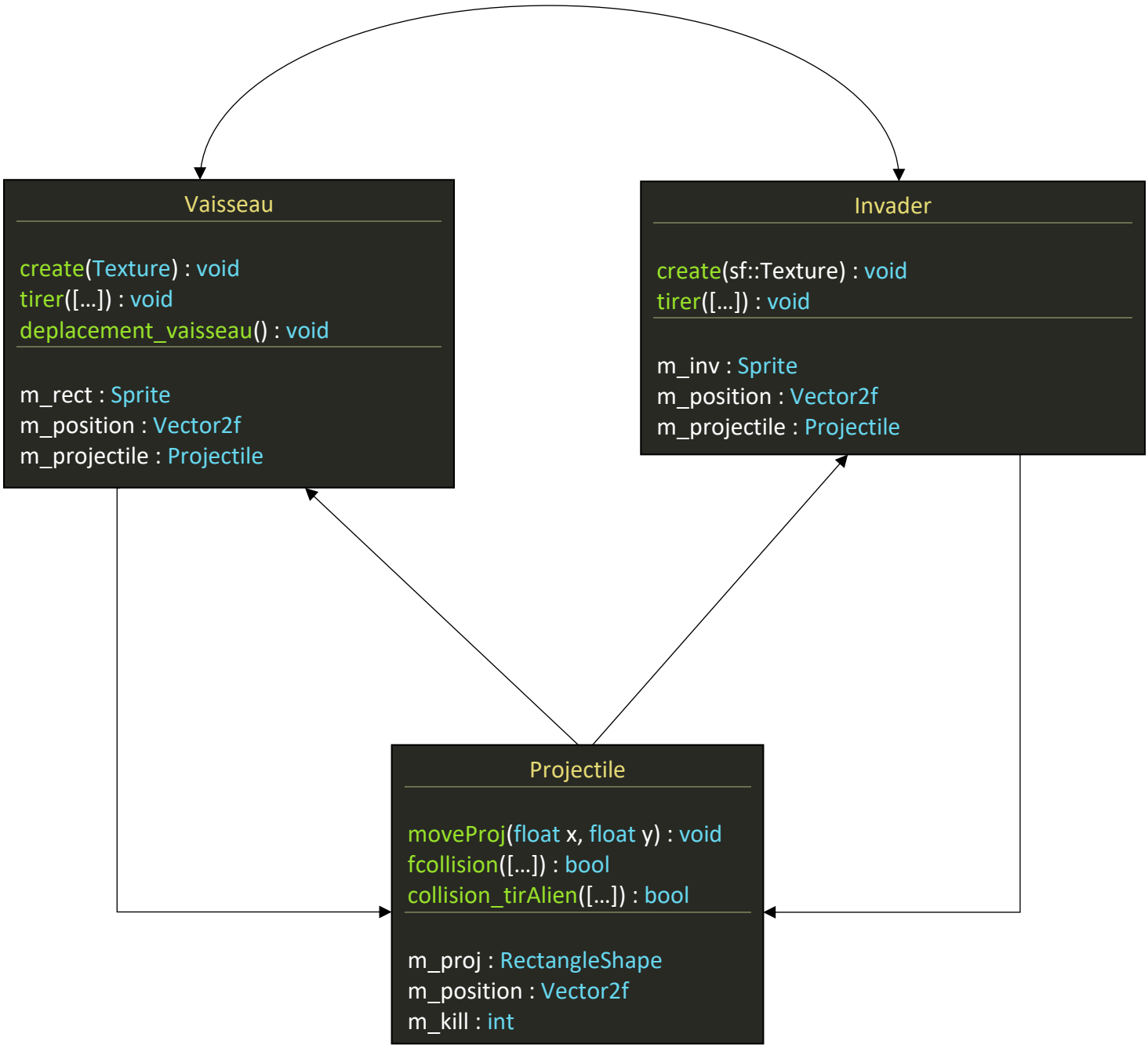
```
class Projectile
{
    /*Déclaration de la classe Projectile qui représentera les projectiles (comme son nom l'indique).
    C'est également elle qui gèrera les collisions et le nombre d'aliens touchés.
        Ce qui est affiché à l'écran est son attribut "m_proj", qui est un sprite*/
public: //Méthodes
    Projectile();
    void setProjPosition(sf::Vector2f);
    void resetKill();
    void moveProj(float x, float y);
    sf::RectangleShape getProj() const;
    int getProjw() const;
    int getProjh() const;
    bool fcollision(Invader tabInv[11][5], int tailleI, int tailleJ,
        Vaisseau &player, sf::Sound& s_explosion, float *sptr);
    bool collision_tirAlien(Vaisseau *player, sf::Sound& s_mort);

private: //Attributs
    int m_width;
    int m_height;
    int m_kill;
    sf::RectangleShape m_proj;
    sf::Vector2f m_position;
};
```

Déclaration de la classe Projectile (fichier Projectile.h)

Les principales méthodes de cette classe sont « fcollision », qui gère les collisions entre les projectiles du joueur et les aliens, et « collision_tirAlien », qui fait la même chose entre les projectiles aliens et le joueur. Les seules instances de cette classe sont dans les attributs des deux classes précédentes. A chaque fois qu'un joueur ou alien tire, la position de son projectile est définie à celle du tireur, puis la méthode tirer est appelée, cette méthode appelant elle-même la méthode de collision adéquate (« fcollision » si c'est le joueur qui tire, « collision_tirAlien » si c'est un alien).

En conclusion, tout ce qui est représenté à l'écran (excepté les menus d'accueil et l'affichage du score et des vies) est déclaré via ces 3 classes.

La figure suivante résume les interactions entre les classes (les flèches représentant les inclusions) :


Vaisseau
create(Texture) : void
tirer([...]) : void
deplacement_vaisseau() : void
m_rect : Sprite
m_position : Vector2f
m_projectile : Projectile
Invader
create(sf::Texture) : void
tirer([...]) : void
m_inv : Sprite
m_position : Vector2f
m_projectile : Projectile
Projectile
moveProj(float x, float y) : void
fcollision([...]) : bool
collision_tirAlien([...]) : bool
m_proj : RectangleShape
m_position : Vector2f
m_kill : int

b) Menus et structure du programme principal

Tout d'abord, le programme principal utilise beaucoup de variables globales car celles-ci sont appelées dans presque toutes les fonctions du programme. L'objet player de la classe Vaisseau, les aliens, les textures, bruitages, etc. font partie de ces variables globales.

Ensuite, la fonction main s'occupe de déclarer certaines variables (des booléens représentants les différents écrans à afficher, la vitesse des aliens, certaines textures et sprites, etc.) puis d'initialiser les textures, sprite et les différentes variables pour l'audio.
Ensuite, une boucle qui se répète tant que la fenêtre est ouverte se lance, et s'occupe d'appeler certaines fonctions du programme nécessaires, ainsi que de gérer quel menu doit être affiché (écran titre, jeu ou information).

Les menus sont choisis grâce à des booléens : en fonction de l'action du joueur un des trois booléens est mis à *true* et les autres à *false*, puis des structures conditionnelles gèrent l'affichage des menus.

IV- Développement

A) Création des classes

La Programmation Orientée Objet étant un domaine qui m'était totalement inconnu au début de ce projet, j'ai donc passé énormément de temps à apprendre et me renseigner sur cette manière de coder, et sur l'utilisation des classes. J'ai donc également eu de nombreuses difficultés lors de cette partie du projet.

a) Vaisseau

La classe à laquelle je me suis attaquée en premier fut la classe Vaisseau, car c'était celle qui me paraissait à la fois la plus importante et la plus simple, et donc plus pratique pour commencer. Mon objectif était de créer une classe qui pourrait non seulement représenter le vaisseau du joueur à l'écran, mais également gérer toute une panoplie de fonctions et variables propres au vaisseau, particulièrement le déplacement, le tir, la gestion du score et des vies.

Après avoir déclaré la classe ainsi que ses méthodes et attributs dans le fichier header, j'ai commencé par coder la fonction « create », qui avait pour but d'initialiser la texture et la position du Sprite du vaisseau, puis la fonction « déplacement », qui ne m'a causée aucun problème.

Juste avant de coder toutes les méthodes « secondaires » (accesseurs et mutateurs) je me suis attaqué à la fonction « tirer », qui fut un peu plus compliquée que prévu, non pas pour sa manière de fonctionner qui est en réalité assez simple : la fonction fait monter le projectile en ligne droite puis appelle la méthode booléenne « fcollision » de l'attribut m_projectile, mais plutôt car elle m'obligeait justement à me mettre à coder la classe Projectile en même temps.

b) Projectile

Je me suis donc mis à réaliser la classe Projectile, qui devait représenter à l'écran le rectangle servant de projectile, mais aussi gérer les collisions de ce dernier.

Comme pour la classe Vaisseau, une fois la classe et ses méthodes et attributs déclarés dans le fichier header, j'ai fait la fonction « fcollision », qui est appelée dans la méthode « tirer » pour l'attribut m_projectile de la classe Vaisseau. Cette méthode m'a posé de nombreux problèmes d'inclusions : en effet je devais à la fois inclure la classe Vaisseau avant la déclaration de Projectile et inclure Projectile avant la déclaration de Vaisseau. Cette double inclusion présentait évidemment des problèmes, et après de nombreuses recherches j'ai opté pour une forward declaration - à la place d'un « include » - de la classe Vaisseau avant la déclaration de la classe Projectile, puisque cette dernière n'utilisait que des références à Vaisseau :

```
class Vaisseau; //On fait une forward declaration de la classe Invader

class Projectile
{
```

Fichier « Projectile.h »

```
2   #include "Vaisseau.h"
3   #include "Vaisseau.cpp"
```

Fichier « Projectile.cpp »

Sur la dernière image on remarque une inclusion de « Vaisseau.cpp » dans le fichier « Projectile.cpp » chose qu'il ne faut normalement jamais faire. Cependant, une erreur d' « undefined reference » vers les méthodes de Vaisseau apparaissait sans cette inclusion. Je me suis donc résigné à l'utiliser, faute d'autre solution.

Une fois les problèmes d'inclusion mutuelle arrangés, j'ai donc pu faire la fonction « fcollision » correctement. La fonction prend en paramètre entre autre le tableau d'Invader (classe qui n'est pas encore créée à ce moment), les dimensions de ce dernier et une référence vers la classe Vaisseau ; et vérifie pour chaque élément du tableau bidimensionnel d'Invader si la position du projectile correspond à celle de l'élément (donc de l'objet Invader) en question. Si c'est le cas, elle actualise le score est la valeur de « m_kill » (nombre d'alien tués) et renvoie *true*. Sinon, elle vérifie si le projectile est hors de l'écran, et renvoie *true* si c'est le cas, *false* sinon.

J'ai ensuite fait la méthode « collision_tirAlien », qui gère les collisions entre les projectiles aliens et le joueur. Comme pour la méthode « fcollision », elle vérifie si la position du projectile de l'alien correspond à celle du joueur (elle prend celui-ci comme paramètre en pointeur). Si c'est le cas, elle réinitialise la position du joueur, lui enlève une vie et renvoie *true*. Sinon elle renvoie *true* si le projectile est sorti de l'écran, *false* sinon.

c) Invader

Une fois les deux classes précédentes terminées, j'ai réalisé le code de la dernière classe : Invader. Cette classe fut relativement facile à faire, étant donné que j'avais déjà rencontrés la plupart des problèmes possibles avec les deux précédentes et que je savais donc comment les résoudre, et que de plus les méthodes de cette classe sont très simples.

En effet, la seule méthode qui n'est ni un accesseur ni un mutateur est la méthode « tirer », qui est identique à la méthode « tirer » de la classe Vaisseau, avec seulement les paramètres pris en entrée qui sont différents.

A noter que cette classe est incluse dans les deux autres classes également, par forward declaration, puisqu'elle est utilisée dans la méthode « fcollision » de la classe Projectile

B) Programme Principal

Le développement du programme principal, dans le fichier « main.cpp », s'est déroulé tout au long du projet en parallèle avec le développement des classes. 7 fonctions en plus de la fonction main sont déclarées dans cette partie, ainsi qu'un grand nombre de variables.

```
/*Fichier contenant les signatures des fonctions du fichier main.cpp*/
void display(bool tir, bool tir_alien);
void createInvaders(float *sptr);
void deplacementInvader(bool droite, float speed);
bool sensDeplacement(bool droite, float *speed);
bool perdu();
void replay();
void jouer(bool *dptr, float *sptr, bool *tir, bool *tir_alien);
```

Fonctions déclarées dans le fichier fonction.h

La fenêtre est déclarée comme variable globale, puis créée au début de la fonction main avec une dimension de 1000x700.

Dans une boucle qui se répète tant que la fenêtre est ouverte, le programme affiche l'écran titre, et lance le jeu si le joueur clique sur start.

En jeu, les seules fonctions appelées dans la boucle sont « jouer » qui, comme son nom l'indique, gère le programme quand il est en mode « jeu », et « display » qui s'occupe d'actualiser et d'afficher les éléments principaux de la fenêtre (joueur, aliens, score et vies, projectiles si le booléen « tir » ou « tir_alien » sont vrais).

La fonction « jouer », quant à elle, appelle la fonction « deplacementInvader » et la méthode « player.deplacement_vaisseau » pour gérer les déplacements.

Ensuite, elle gère les tirs avec le fonctionnement suivant : si une première condition est vérifiée (touche espace pressée pour le tir joueur, un certain temps aléatoire atteint pour le tir alien) le booléen de tir correspondant est mis à *true*, puis dans une 2^{e} condition si ce booléen est vérifié on appelle la méthode de tir du joueur ou de l'alien. Le booléen de tir est remis à *false* si la méthode de collision du projectile correspondante renvoie *true.*

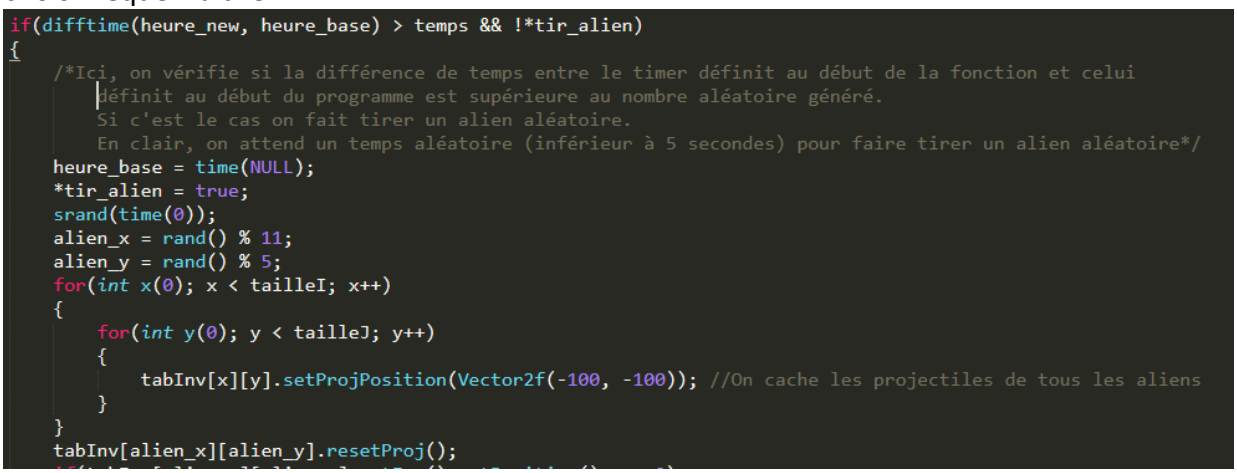
Le système d'aléatoire pour le tir ennemi est géré de la manière suivante : une première variable time_t (appartenant au module time.h) est intialisée à l'heure du système au début du programme. Une deuxième variable time_t est initialisée à l'heure du système à chaque tour de la fonction « jouer », ainsi qu'un nombre aléatoire compris entre 0 et 5.

```
void jouer(bool *dptr, float *sptr, bool *tir, bool *tir_alien)
{
    /*Fonction principale qui gère le système de jeu*/
    heure_new = time(NULL); //On intialise une 2e variable temps qui prend l'heure du système comme valeur
    double temps;
    srand(time(0));
    temps = rand() % 5; //On prend un nombre aléatoire entre 0 et 5
```

Début de la fonction « jouer », initialisation d la variable de temps et de nombre aléatoire

Ensuite, dans une des conditions de tir on vérifie si la différence de temps entre les deux variables est supérieure à la variable aléatoire « temps ». Si c'est le cas, on actualise la

première variable de temps, et on prend 2 coordonnées aléatoire dans le tableau d'alien pour choisir lequel va tirer :



```
if(difftime(heure_new, heure_base) > temps && !*tir_alien)
{
    /*Ici, on vérifie si la différence de temps entre le timer définit au début de la fonction et celui
        définit au début du programme est supérieure au nombre aléatoire généré.
        Si c'est le cas on fait tirer un alien aléatoire.
        En clair, on attend un temps aléatoire (inférieur à 5 secondes) pour faire tirer un alien aléatoire*/
    heure_base = time(NULL);
    *tir_alien = true;
    srand(time(0));
    alien_x = rand() % 11;
    alien_y = rand() % 5;
    for(int x(0); x < tailleI; x++)
    {
        for(int y(0); y < tailleJ; y++)
        {
            tabInv[x][y].setProjPosition(Vector2f(-100, -100)); //On cache les projectiles de tous les aliens
        }
    }
    tabInv[alien_x][alien_y].resetProj();
```

Début de la condition qui vérifie la différence de temps entre les 2 variables de temps

La fonction « difftime » est une fonction appartenant au module « time.h ».

Une fois les tirs gérés, la fonction « jouer » vérifie si la fonction « perdu » renvoie *true*, et appelle la fonction « replay » si c'est le cas. Celle-ci se charge d'afficher un écran de Game Over, et renvoie sur l'écran titre si n'importe quelle touche est pressée.

V- Manuel d'utilisation

A) Lancement de l'application

Actuellement, je n'ai pas encore réussi à rendre l'exécutable de l'application fonctionnel, aussi le seul moyen de la lancer est via Code::Blocks.
Quand on lance l'application, un écran titre s'affiche :

Ecran titre

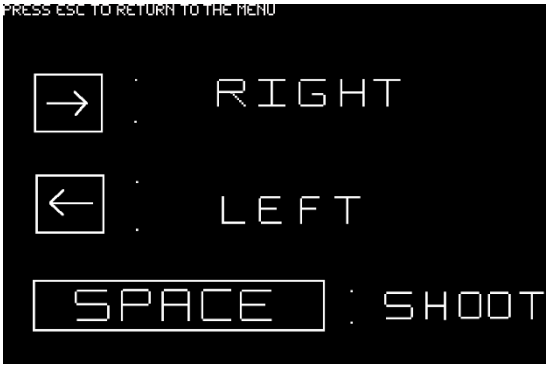
Sur cet écran titre, on a le choix entre trois bouton : start, informations et quit. Comme leurs noms l'indiquent, ces boutons permettent de commencer une partie, avoir des informations sur comment jouer et quitter le jeu.

B) Comment jouer

Une fois qu'une partie est lancée en cliquant sur start, on arrive sur l'écran de jeu :

Ecran de jeu

La manière de jouer est simple : le joueur peut se déplacer à droite ou à gauche avec les flèches directionnelles correspondantes, et tirer un projectile en appuyant sur la barre d'espace. Cependant, il ne peut tirer qu'une fois le projectile précèdent sorti de l'écran ou ayant touché un ennemi.



Le menu information

L'objectif est de toucher un maximum d'alien avec ses projectiles, tout en évitant les projectiles ennemis tirés de manière aléatoire.

Chaque alien rapporte un certain nombre de point : Ceux des 2 premières lignes du bas rapportent 10 points, ceux de la troisième et quatrième ligne rapportent 20 points et enfin ceux de la dernière ligne (la plus haute) rapportent 40 points.

Le joueur peut revenir à l'écran titre à tout moment en appuyant sur la touche echap.

C) Fin de partie

Une partie se termine si le joueur s'est fait toucher trois fois par des projectiles ennemis ou si un alien atteint le bas de l'écran. Quand le joueur perd, l'écran de Game Over suivant s'affiche :

Ecran de fin de partie

Lorsque cet écran s'affiche, il reste affiché jusqu'à ce que le joueur appuie sur une touche du clavier. Alors, l'écran titre s'affiche de nouveau et le joueur peut relancer une partie. S'il le fait, son score et son nombre de vies sont réinitialisés, il repart donc de zéro.

VI- Conclusions

A) Perspectives

Tout d'abord, plusieurs éléments auraient pu être ajoutés au cahier des charges. En effet, nous avions plusieurs idées que j'aurai pu intégrer au jeu avec plus de temps : un système de bonus (tirer plus vite, être invincible pendant un certain temps, etc.), mettre des obstacles entre le joueur et les ennemis, ajouter des « boss » (des ennemis plus robuste, apparaissant au bout d'un certain temps), un menu option, etc.

De plus, étant donnée mon inexpérience en la matière, mon organisation des classes est quelque peu bancale, avec par exemple une triple inclusion mutuelle, et surtout l'inclusion du fichier Vaisseau.cpp dans Projectile.cpp, qui est une erreur. Aussi, les attributs et méthodes de mes classes auraient pu être organisés de meilleure manière.

B) Conclusions

a) Fonctionnement de l'application

En conclusion, quelques bonnes idées n'ont pas pu être réalisées, comme dit dans la partie précédente, et beaucoup de choses auraient pu être améliorées à tous les niveaux, que ce soit sur les graphismes, musique et bruitage, ou même le jeu et le code en lui-même.
De plus, pour l'instant on ne peut pas utiliser l'exécutable du programme à cause de plusieurs erreurs au démarrage que je n'ai pas réussi à résoudre (entre autre, une fenêtre d'erreur annonçant « l'application n'a pas réussi à démarrer correctement (0xc000007b) »). Pour lancer le jeu, il faut donc utiliser une version de Code::Blocks et SFML 2.5.1, il faut également que Code::Blocks soit correctement configuré pour utiliser SFML.

Cependant, j'ai été capable de coder un jeu sans aucun bug (sur mon ordinateur du moins), tout en utilisant beaucoup de nouvelles notions que je ne connaissais pas : l'utilisation de classes, la programmation orientée objet ainsi que la bibliothèque graphique SFML. De plus, le jeu répond aux attentes du cahier des charges.

b) Fonctionnement du groupe de travail

Malheureusement, la cohésion du groupe était inexistante étant donné que mon binôme (Thomas Faure) a décidé d'abandonner la formation en cours de route. Je me suis donc occupé seul de l'intégralité du projet, excepté pour les graphismes et bruitages dont s'est occupé Thomas.

Cependant, même en étant seul, j'ai su efficacement gérer le temps que j'avais pour avancer à un bon rythme, et j'ai également su faire face aux difficultés que représentait un projet de cet ampleur, surtout seul.

Pour conclure ce rapport, malgré un groupe qui n'a pas fonctionné, j'ai su réaliser seul un projet qui m'a permis d'apprendre non seulement beaucoup de choses sur le codage en général, mais également sur la gestion d'un projet dans son ensemble. Au final, j'ai répondu à toutes les attentes du cahier des charges en codant entièrement un jeu fonctionnnel.