

POLYTECH SORBONNE

EPU-N7-IOB

---

## Projet final : Space Invader v2

---



Arthur Guillec

# 1 Introduction

Ce rapport est joint au rendu du projet final de l'UE EPU-N7-IOB dont le thème est "Le monde d'après".

Le programme est un jeu 2D où le joueur se livre à une guerre spatiale dans un univers futuriste.

Nous traiterons des principaux aspects du programme et de ses sources.

## 2 Le jeu

L'objectif est similaire à celui du jeu **Space Invaders**. Le joueur est aux commandes d'un vaisseau spatial et doit se défendre face à une horde d'ennemis déterminés.

Il est possible de se déplacer dans toutes les directions du plan 2D. A chaque fois qu'un vaisseau ennemi est détruit, le score du joueur s'incrémente de un. L'objectif est d'avoir le plus grand score à la fin de la partie, i.e lorsque le vaisseau du joueur n'a plus de vie.

### 2.1 Dépendances

La partie audio et graphique du programme nécessite la bibliothèque **SFML** (Simple and Fast Multimedia Library). Elle s'installe facilement sur Debian et ses dérivés via la commande :

```
$ sudo apt-get install libsfml-dev
```

### 2.2 Compilation

Les sources du programme peuvent être récupérés depuis le dépôt **GIT** via la commande

```
$ git clone https://github.com/3700240/guillec-projet-cpp
```

Un Makefile contenant une règle **all** et **clean** est fourni. Une fois la bibliothèque **SFML** installée, il ne reste donc plus qu'à exécuter la commande

```
$ make
```

depuis le répertoire guillec-projet-cpp. puis

```
$ ./game.bin
```

pour lancer une partie. Bon jeu !

## 3 Les classes

Le diagramme UML associé à la section qui va suivre est disponible dans le même répertoire que ce rapport sous le nom **diagramme.svg**. Il a été décidé de ne pas directement l'intégrer au rapport pour que le lecteur puisse profiter pleinement du format vectoriel (zoomer comme bon lui semble).

### 3.1 Entity

La classe **Entity** est la «super» classe mère dont dérivent toutes les entités, i.e les différents types de vaisseaux et de projectiles.

```
class Entity : public sf::Drawable, sf::Transformable
```

Elle hérite de `sf::Drawable` et `sf::Transformable`. Toutes ces classes filles doivent donc définir la méthode

```
virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
```

Cela permet de les dessiner simplement avec la méthode **draw** de `sf::RenderWindow`

```
// exemple
Entity foo;
...
_win.clear();
_win.draw(foo);
_win.display();
```

Chaque entité possède une position, une direction, une vitesse, un rayon (les collisions sont approximées par des cercles).

De plus, elles possèdent un attribut un peu particulier : **isAlive**. Ce dernier décrit si une entité est toujours active. Si ce n'est pas (plus!) le cas, elle sera supprimée à la prochaine itération par des mécanismes similaires à un garbage collector.

### 3.2 Character

La classe **Character** représente un vaisseau quelconque. Un vaisseau peut être détruit et possède donc une vie. Il est nécessaire d'avoir cette classe entre **Entity** et **Player/Enemy** car les projectiles font des dégâts à tous les vaisseaux, i.e à des **Character**.

Toutes les classes filles de **Character** doivent définir la méthode

```
virtual bool allowedToFire()=0;
```

Cette méthode renvoie **true** si à une itération quelconque de la boucle de jeu, le **Character** peut tirer une balle.

### 3.3 Player

La classe **Player** représente le vaisseau du joueur. Un joueur peut tirer une balle toutes les 10 itérations, soit 6 fois par seconde (la logique a lieu à pas constant de 1/60s)

```
void Player::goTo(sf::Vector2f targetPos)
{
    sf::Vector2f relTargetPos = targetPos-_pos;

    float dot = _dir.x*relTargetPos.x + _dir.y*relTargetPos.y;
    float det = _dir.x*relTargetPos.y - _dir.y*relTargetPos.x;

    float angle = toDegree(std::atan2(det, dot));
    float anglemax = 4.f + (400.f-_speed)/300.f*3.f;
    float rot = (angle>0) ? std::min(angle, anglemax) : std::max(angle, -anglemax);

    _dir = sf::Vector2f(std::cos(toRadian(rot))*_dir.x-std::sin(toRadian(rot))*_dir.y,
        std::sin(toRadian(rot))*_dir.x+std::cos(toRadian(rot))*_dir.y);
    _speed = std::max(std::min(2*magnitude(relTargetPos),400.f) ,100.f);
}
```

Sans entrer dans les détails trigonométriques, le joueur change de direction et de vitesse à chaque itération. La nouvelle direction dépend de la position de la souris sur l'écran et ne peut être trop différente de la précédente (la ligne `rot=...` s'assure que l'angle entre l'ancienne direction et la nouvelle n'est pas trop grand).

La vitesse dépend de la distance entre la souris et le joueur. Plus cette dernière est grande, plus la vitesse l'est aussi.

### 3.4 Enemy

La classe **Enemy** est très similaire à **Player**. Elle a les mêmes méthodes. La différence est que les actions des **Enemy** ne dépendent d'aucune entrée utilisateur. Ce sont donc des I.A. Par exemple, l'action de tir est régit par de l'aléatoire. De plus, la vitesse des ennemis est constante.

### 3.5 Projectile

La classe **Projectile** représente tout objet susceptible de causer des dégâts et qui peut être lancé depuis un vaisseau.

Contrairement aux **Character**, les projectiles n'ont pas un attribut de vie (**\_health**) mais un attribut de dégâts nommé **\_damage**. Il représente les dégâts qu'inflige un projectile si il touche une cible.

### 3.6 Bullet

La classe **Bullet** est (pour l'instant ?) l'unique classe fille de **Projectile**. Elle représente une balle à la trajectoire rectiligne uniforme, à même d'infliger des dégâts aux différents types de vaisseaux (i.e **Character**).

Lorsqu'une balle touche une cible, elle lui inflige des dégâts puis est supprimée. De la même façon, une balle qui sort de l'écran est supprimée.

### 3.7 ResourceManager

Le template **ResourceManager** a pour rôle de rendre plus simple la manipulation des ressources audios et graphiques. Il s'agit essentiellement d'un **std::map** aux fonctionnalités étendues. Voici les principales méthodes :

```
template <typename Resource, typename Identifier>
class ResourceManager
{
public:
    void load(Identifier id, const std::string& filename);
    Resource& get(Identifier id);
    const Resource& get(Identifier id) const;
private:
    void insertResource(Identifier id, std::unique_ptr<Resource> resource);
private:
    std::map<Identifier, std::unique_ptr<Resource>> _resourceMap;
};
```

Prenons par exemple la classe **Enemy** qui permet d'instancier les différents vaisseaux ennemis. Cette classe contient un attribut privé **\_sprite** de type **sf::Sprite** utilisé par SFML pour l'affichage. Cet attribut a besoin d'une texture pour être initialisé. Comme il serait coûteux de lire le fichier **enemy.png** associé à cette texture à chaque instanciation d'ennemis, on stocke au démarrage du programme la texture dans un **TextureManager** (i.e un **ResourceManager**<**sf::Texture**, **Textures::ID**> et on y accède via la méthode **get** et l'identifiant approprié.

```
_sprite(textures.get(Textures::Enemy))
```

On définit donc trois types de **ResourceManager** : un pour chacun des types de ressources manipulées.

```
typedef ResourceManager<sf::Texture, Textures::ID> TextureManager;
typedef ResourceManager<sf::Font, Fonts::ID> FontManager;
typedef ResourceManager<sf::SoundBuffer, Sounds::ID> SoundManager;
```

### 3.8 GameInstance

La classe **GameInstance** encapsule l'ensemble des données relatives à une partie.

```
private:
    sf::RenderWindow _win;

    TextureManager _textures;
    FontManager _fonts;
    SoundManager _sounds;

    Player* _player;
    std::list<Enemy *> _enemyList;
    std::list<Bullet *> _bulletList;

    sf::Sprite _background;
    sf::Sound _sound;
    sf::Music _music;
    int _score;

    static const sf::Time _frameduration;
```

On retrouve notamment la fenêtre de rendu **\_win**, trois **ResourceManager** pour chacun des types de ressources manipulés (**sf::Texture**, **sf::Font** et **sf::Sound**), les listes dans lesquelles sont stockées les ennemis et les balles ainsi qu'un pointeur vers le **Player**.

Les méthodes de cette classe sont en charge de la logique du jeu. C'est d'ailleurs dans la méthode **run()** que se trouve la boucle de jeu.

## 4 Boucle de jeu

Le jeu est articulé autour d'une boucle infinie qui traite les événements (entrées de l'utilisateur), met à jour le jeu (déplacer les entités, gérer les collisions, etc.) puis affiche l'état actuel.

Une des principales difficultés est de faire en sorte que la logique du programme se fasse à fréquence constante tandis que l’affichage se fasse aussi vite que possible.

En effet, si on travaille à pas de temps variable dans la logique de jeu, le risque est que si pour une raison quelconque le programme ralentit, le pas de temps va exploser et nous rencontrerons des problèmes dans les fonctions de collisions.

```
void GameInstance::run()
{
    sf::Clock clock;
    sf::Time timeSinceLastUpdate = sf::Time::Zero;
    while(_win.isOpen())
    {
        sf::Time dt = clock.restart();
        timeSinceLastUpdate += dt;
        while(timeSinceLastUpdate > _frameduration)
        {
            timeSinceLastUpdate -= _frameduration;
            manageNewEvents();
            if(_player->isAlive())
                update(_frameduration);
        }
        render();
    }
}
```

L’idée dans cette boucle est que la logique est mise à jour uniquement si le temps écoulé depuis la dernière màj est supérieur au pas de temps choisi (ici **\_frameduration**). L’affichage lui est fait à chaque itération. Ainsi, on obtient bien le comportement souhaité d’un affichage aussi rapide que possible et d’une logique à pas de temps constant.

## 5 Collisions

Les collisions ont été simplifiées au maximum. Le masque de collision des entités est approximé par un cercle.

Prenons par exemple la méthode **updateEnemies()**. En plus de mettre à jour la position des ennemis et de supprimer ceux qui sont morts, elle vérifie que l’ennemi n’est pas entré en collision avec le joueur.

```
void GameInstance::updateEnemies()
{
    // Pour chaque ennemi
```

```

for (std::list<Enemy*>::iterator it=_enemyList.begin(); it!=_enemyList.end(); ++it)
{
    Enemy *e = *it;
    if(e->isAlive())
    {
        e->goTo(_player->getPos()); // On change la direction de l'ennemi
        e->update(_frameduration); // On met à jour sa position

        // Gestion des collisions
        if(distance(e->getPos(), _player->getPos()) < e->getRadius()+_player->getRadius())
        {
            _player->takeDamage(40); // Le joueur prend des dégâts
            e->kill(); // L'ennemi meurt
        }
    }
    else // Garbage collector
    {
        _score++;
        delete(e);
        it = _enemyList.erase(it);
    }
}
}

```

Les collisions entre ennemis n'ont volontairement pas été implémentées. En effet, le jeu serait un peu trop simple sinon !

## 6 Tests unitaires

Les tests unitaires ont été réalisés avec la bibliothèque `<cassert>`. On les retrouve principalement dans les parties critiques du code.

Un exemple est la fonction **normalized**

```

sf::Vector2f normalized(sf::Vector2f vector)
{
    assert(vector != sf::Vector2f(0.f, 0.f));
    return vector / magnitude(vector);
}

```

Il serait très dangereux de calculer le vecteur unitaire d'un vecteur nul (division par zéro). On utilise donc un test unitaire pour vérifier le bon comportement du programme.



## 7 Conclusion

Ce projet fut l'occasion d'aborder la POO dans le contexte du développement de jeu vidéo et d'apprendre à utiliser des bibliothèques graphiques telles que la SFML.

Cela a été très enrichissant car de tels programme obligent à se poser des questions sur la conception et usent pleinement des fonctionnalités avancées du C++ 11 (et au delà).