Projet C++ : Yellow

YUN VS The Yellow K-Pop Gang

Dimitri KOKKONIS, Victor VERBEKE

Année scolaire 2018 - 2019
EI-SE 4

Introduction

L'application développée, "YUN VS The Yellow K-pop Gang", est un jeu dit "Shoot'em Up" ("abattez-les" en français). Semblables à Space Invaders, ces jeux contiennent généralement un personnage qui doit tuer plein d'ennemis en survivant aux vagues d'ennemis.

Dans le cadre de notre jeu, le scénario est simple : Yun, élève en EI-SE 4, déteste la K-pop et a décidé d'aller vaincre les stars du Yellow K-pop Gang afin de se sentir apaisé. La blague vient du fait que Yun déteste réellement la K-pop (un genre de musique spécifiquement coréen), et qu'on aurait trouvé très drôle de faire un jeu sur ce sujet.

Le jeu est divisé en trois niveaux, chacun présentant des vagues d'ennemis et un boss. Chaque boss et chaque ennemi est une star de K-pop (pour les niveaux 1, 2 et 3, les boss sont respectivement Beenzino, un rappeur coréen, une membre du groupe d'idols Girl's Generation, et Jonghyun, une star de la scène coréenne).

Application des contraintes

Le jeu remplit donc les contraintes suivantes :

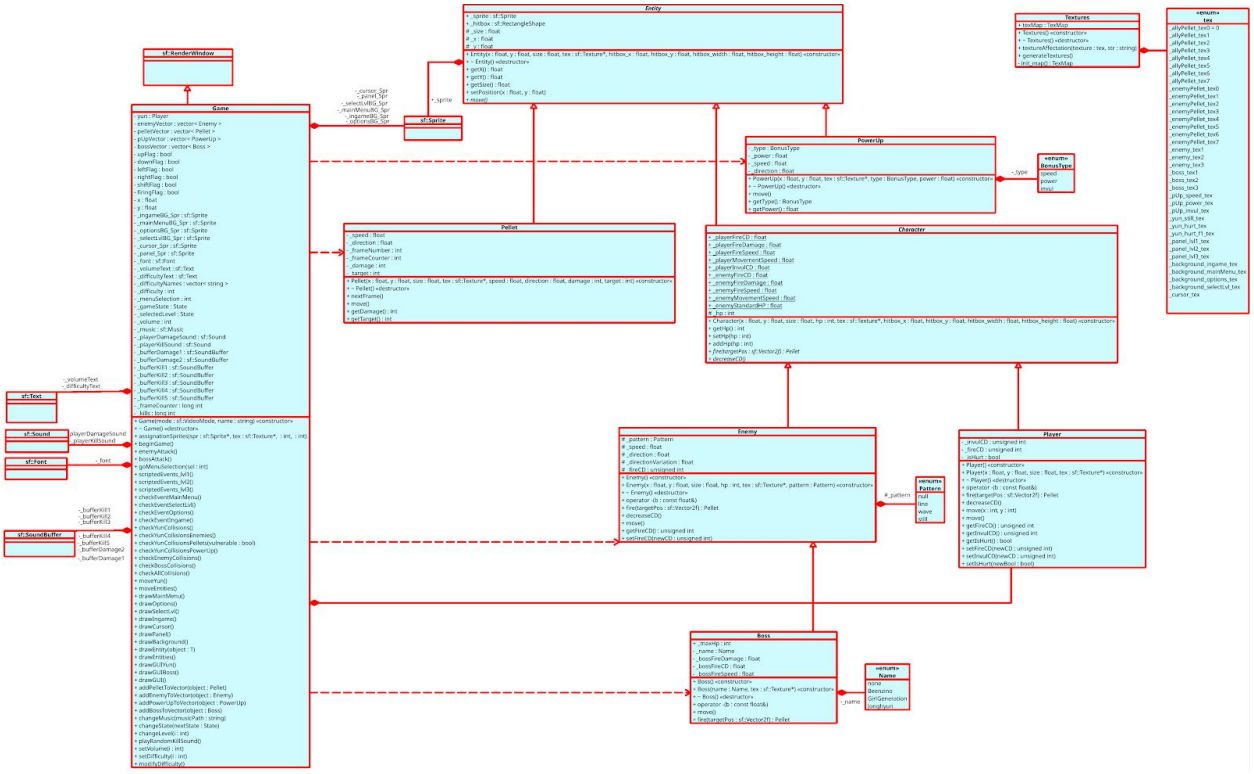
- Huit classes (Game, Textures, Entity, Character, Player, Enemy, Boss, PowerUp et Pellets, soit neuf classes au total).
- Trois niveaux de hiérarchie (Boss hérite d'Enemy, qui hérite de Character, qui hérite de Entity, soit quatre niveaux).
- Deux fonctions virtuelles (Entity et Character sont des classes virtuelles, chaque Entity peut move() et setPosition(), tandis que chaque Character peut tirer avec fire() et réduire les cooldowns de tir avec decreaseCD(), soit quatre fonctions virtuelles).
- Deux surcharges d'opérateurs (Player, Enemy et Boss ont un opérateur void operator-(const float &b) pour prendre des dégats et faire d'autres choses, comme changer des sprites pour Player, donc trois surcharges d'opérateur).
- Deux conteneurs de la STL (on store toutes les sf::Texture dans un map dans la classe Textures, et chaque Pellet, Enemy, PowerUp et Boss est stocké dans des vecteurs associés).
- Chaque fichier est commenté, notamment pour la classe Game où se déroule la majorité du code.
- Le diagramme UML est plus bas.
- Le dépot git est le suivant : https://github.com/VictorVerbeke/yellow
- Le Makefile est réglé correctement, avec les règles all et clean.

Cependant, certaines contraintes ne sont pas respectées :

- Le code présente BEAUCOUP d'erreurs avec Valgrind. Nous pensons que cela est dû à la bibliothèque SFML, car nous avons effectué des tests (cf. learning et learning2 du dépot git) et nous avons quand même observé une grande quantité d'erreurs. La bibliothèque semble donc fuir autant qu'une passoire, et comme nous avons correctement effectué la POO (presque aucun pointeur, que des retour d'objet, les seuls pointeurs étant les sf::Texture car sinon on obtenait des sprites complètement blancs), nous pensons que la grande majorité vient de la librairie.
- Certaines fonctions font plus de trente lignes, mais restent simple à lire.

UML de l'application

Lien de l'image haute définition : https://imgur.com/bjDXMUX



Code et Architecture de POO

Le jeu est fait en C++ et a subi plusieures réécritures durant son développement. Au départ, Victor n'étant pas réellement familier avec le C++, il a fait plein de pointeurs (globalement, toutes les fonctions retournaient des pointeurs, demandaient en argument des pointeurs, chaque vecteur était un vecteur de pointeur) et pas vraiment orienté objet (toutes les fonctions presque étaient dans la classe Game qui modifiait l'emplacement de toutes les entités au lieu d'appeler les méthodes de déplacement des entités).

Après avoir compris que le C++, c'est pour éviter les pointeurs et manipuler des objets, il a réécrit le code pour obtenir un résultat plus orienté objet.

Le jeu tourne principalement autour de la classe Game, qui est grossièrement expliqué, l'administrateur du programme qui régit qui fait quoi. Dans Game sont stockés des méthodes et des attributs pour gérer les objets et les faire interagir entre eux. Par exemple, on peut observer les méthodes suivantes :

```
// Methodes de collision
void checkYunCollisions();          // Appelle les fonctions ci-dessous
void checkYunCollisionsEnemies();   // pour les type d'objet avec
void checkYunCollisionsPellets(bool vulnerable); // lesquels Yun peut
void checkYunCollisionsPowerUp();   // interagir.
void checkEnemyCollisions();    // Check les collisions des ennemis
void checkBossCollisions();     // Les boss aussi. Pas de discrimination.
void checkAllCollisions();      // Appelle toutes les détections de
                                // collision.

// Methodes de déplacement
void moveYun();             // Fait bouger le joueur selon les input
void moveEntity(Player object, float x, float y);
```

On a là des méthodes qui permettent de traiter toutes les entités du jeu afin de faire des vérifications, une par une, des entités. Mais pour faire tout ça, il faut bien stocker toutes les entités ! En effet, elles sont là (méthodes d'ajoute et attributs associés) :

```
// Methodes d'ajout d'instances
void addPelletToVector(Pellet object);     // Les entités (sauf Joueur)
void addEnemyToVector(Enemy object);       // sont stockées dans des
void addPowerUpToVector(PowerUp object);   // vecteurs. On les ajoute
void addBossToVector(Boss object);         // grâce aux fonctions ici.
```

```
// Attributs : Entités
Player yun;                     // Ici sont stockées toutes les entités.
vector<Enemy> enemyVector;      // Yun est unique, mais on peut avoir
vector<Pellet> pelletVector;    // plein d'entités à l'écran. Ainsi,
vector<PowerUp> pUpVector;      // on les stocke dans des vecteurs pour
vector<Boss> bossVector;        // les afficher via itérateurs dans les
                                // fonctions appropriées.
```

Donc la classe Game gère les collisions et mouvement des entités. Mais plus que ça, il faut savoir que la classe Game sert surtout pour afficher tout le jeu ! En effet, avant (sur le brouillon) elle s'appelait Window, mais Game c'est plus court et plus représentatif de la globalité des actions de la classe.

```
class Game : public sf::RenderWindow {
```

```
// Methodes d'affichage générales
void drawMainMenu();    // Selon l'état du jeu où l'on est, il faut
void drawOptions();     // afficher différentes choses. Pour chaque
void drawSelectLvl();   // état il y a une fonction courte qui permet
void drawIngame();      // d'afficher les menus ou le jeu en cours.

void drawCursor();      // Permet d'afficher le curseur.
void drawPanel();       // Affiche le panel associé au niveau choisi.
// Methodes d'affichage Ingame
void drawBackground();          // En jeu, on veut afficher le fond,
                                    // puis chaque entité. On a donc une
template<class T>                   // méthode pour chaque entité, puis
void drawEntity (T object);         // une méthode drawEntities pour
void drawEntities();                // toutes les appeler.
```

La classe hérite donc de sf::RenderWindow, et sert à afficher toutes les entités, les background, etc…

Mais comme le disait l'oncle de Jackie Chan, "je n'ai pas fini !". Car en effet, la classe Game, en plus de gérer les entités et d'afficher le jeu, et bien… gère le flow du jeu en lui-même.
Regardez cette image à droite !
En effet, la classe Game permet de gérer le jeu comme ce que l'on appelle en électronique numérique une machine à

```
enum State {beginState = 0,
            mainMenu = 1,
            options = 2,
            selectLvl = 3,
            level1 = 4,
            level2 = 5,
            level3 = 6};
```

état : si le jeu se trouve dans tel état, alors on effectue le comportement associé. On obtient donc les méthodes suivantes :

```
// Methodes du jeu
void beginGame();       // Démarre le jeu, boucle principale.
void enemyAttack();     // Automatise l'attaque des ennemis.
void bossAttack();
void goMenuSelection(int sel);  // Permet de naviguer entre les états
                                // du jeu.

// Methodes de codage des niveaux scriptés.
void scriptedEvents_lvl1(); // Permet l'apparition d'ennemis ingame
void scriptedEvents_lvl2(); // A chaque niveau correspond une série
void scriptedEvents_lvl3(); // d'apparition, donc des méthodes diff.

// Methodes de gestions d'Event
void checkEventMainMenu();  // Selon l'état du jeu (menu principal
void checkEventSelectLvl(); // selection de niveau, menu d'options
void checkEventOptions();   // ou encore en jeu, gestion des events
void checkEventIngame();    // différente.
```

Là-dedans, on trouve beginGame qui gère justement le déroulement du jeu selon l'état du jeu, mais aussi la gestion des attaques des entités, les événements scriptés des niveaux, la navigation dans les menus, les vérifications des inputs, toutes ces fonctions dépendent de l'état du jeu (en jeu ? dans le menu principal ?). De plus, nous avons d'autres méthodes encore (promis c'est presque les dernières) pour gérer l'état du jeu, ainsi que d'autres variables telles que le son et la difficulté.

```
// Gestion du son, des niveaux, de la difficulté
void changeMusic(string musicPath); // Modifie la musique lue
void changeState(State nextState);  // Modifie l'état du jeu.
void changeLevel(int i);    // Choisit le niveau suivant ou précédent
void playRandomKillSound(); // Joue un son de mort au hasard.
void setVolume(int i);      // Change le volume de +/- 5%
void setDifficulty(int i);  // Monte ou baisse la difficulté.
void modifyDifficulty();    // Modifie les attributs statiques des characters.
```

Bref, la classe Game est une super classe (pas dans le sens POO, mais plus en sens de taille) avec un flow assez imposant, et si nous devions être fiers d'un aspect technique du programme, ce serait cette classe. Cependant, ce dont nous sommes réellement fiers, c'est des menus et des effets sonores, qui ont été très drôles à implémenter et nous ont fait rire tout le long du projet.

Concernant les autres objets. Ils dérivent tous d'Entity, qui agit donc de classe abstraite : toutes les autres classes héritent de plus ou moins loin d'Entity (tout comme tous les animaux du monde sont des… animaux…). Voici donc la classe.

```
class Entity {
    public:
        Entity (float x, float y, float size, sf::Texture *tex,
            float hitbox_x, float hitbox_y, float hitbox_width, float hitbox_height);
        ~Entity();

        // Getter, Setter.
        float getX(){ return _x; }
        float getY(){ return _y; }

        // Méthodes virtuelles
        virtual void move() = 0;
        virtual void setPosition(float x, float y);

        float _size;
        float _x;
        float _y;
        sf::Sprite _sprite;
        sf::RectangleShape _hitbox;
    protected:

};
```

Chaque entité possède donc un sprite, une hitbox, une position, une taille.

Dans ses enfants, nous trouvons :

- Pellet, qui est en gros les projectiles du jeu : ils possèdent vitesse, direction, une cible, des dégats et des compteurs pour de l'animation (ce sont les seuls à être animés d'ailleurs, on avait penser animer Player mais on a pas assez d'images).
- PowerUp, qui contient un enum sur le type de bonus apporté, ainsi que sa puissance, et aussi une direction et une vitesse.
- Character, classe abstraite qui est la base de tous les autres objets du jeu, contenant toutes les variables statiques concernant les caractéristiques des ennemis, du joueur, et un système de points de vie.

Les variables statiques de Character permettent de modifier chaque entité à partir du menu de difficulté, qui modifie les variables statiques. Il s'agit donc d'un raccourci (au lieu de modifier les statistiques de chaque variable) très utile.

Dans les enfants de Character, nous trouvons :

- Player, le joueur principal qui reçoit ses déplacements de Game, et qui sait gérer ses cooldowns d'invulnérabilité et de tir, et qui tire.
- Enemy, assez semblable à Player au final sauf que ses méthodes sont un peu différentes, déplacement avec vitesse/direction, et tirs grâce à une position donnée (celle du joueur).
- Boss hérite de Enemy et possède juste un enum Name { … } pour différencier chaque boss, et des variables spéciales pour ne pas utiliser celles définies en statique dans Character. Les caractéristiques de chaque boss sont déterminées par son nom dans son constructeur, afin de rendre chaque boss différent.

Nous avons finalement une dernière classe un peu à part, "Textures". Mais que fait Textures ? Et bien, Textures tourne autour d'un très grand enum, définissant le type tex, qui sert de clé à un map (nous avons défini un ***typedef map<tex, sf::Texture*> TexMap***, et nous avons donc un map de type TexMap).
Les clés permettent d'arriver donc sur des pointeurs sur sf::Texture (pointeurs car sinon on arrivait à des problèmes au niveau du code, les sprites devenaient blancs).
A chaque clé est donc associée une texture, et la classe Texture est instanciée au début du programme. A ce moment-là, toutes les textures sont chargées en jeu. Cela permet d'éviter la création de multiples textures pour des objets semblables (deux ennemis avec la même tête, on évite de load deux fois une texture pour ensuite les utiliser pour deux sprites, et on préfère que les deux sprites soient basés sur la même texture initialisée au début du jeu).

Voilà voilà. Il y a plus de détail spécifiques en commentaire du code, et il y a un README.md aussi. On espère que vous allez apprécier le projet !

Edit : Après avoir fini le rapport, on a mis un système de barres de vie, mais c'est assez semblable aux hitbox.