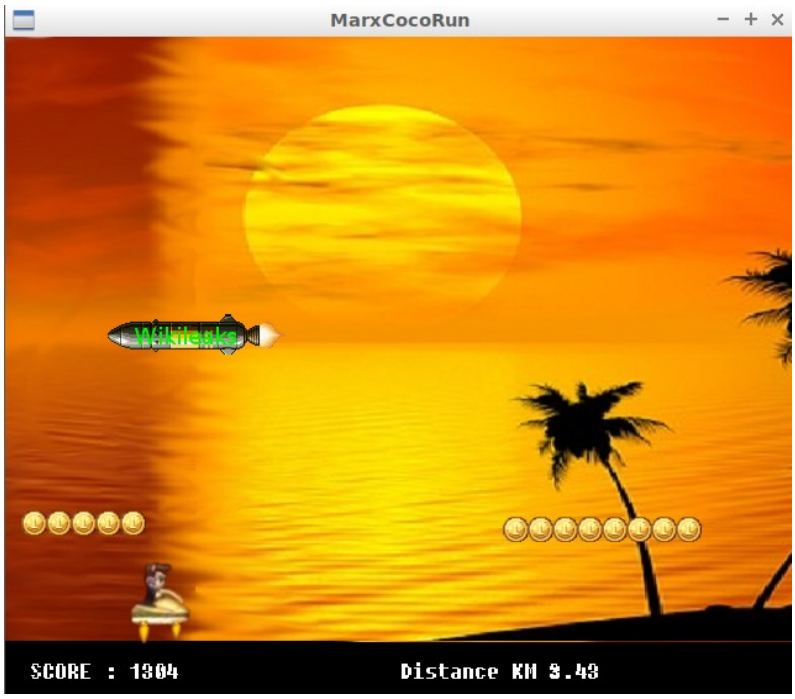
Tristan CLARE
Bao Tinh PIOT

Rapport Projet C++



Électronique Informatique – Systèmes Embarqués Année 4
Année Universitaire 2017 - 2018

I – Introduction

«Merci StackOverflow et le TP sur les atomes et les opérateurs Binaires ! »

A – Présentation du Jeu

MarxCocoRun est un jeu de plate-forme 2D. Vous incarnez Bob, un multi milliardaire qui doit amasser le plus de sous possible tout en évitant les « missiles importunateurs » (missile communiste, la cour pénale, les anges de Dieu et Wikileaks) qui peuvent détruire votre fortune !

Bob ramasse de l'argent à l'aide de son paquebot jet pack plaqué or. Si le joueur appuie sur la touche « ESPACE », le paquebot décolle ; au contraire si le joueur n'appuie plus sur la touche espace, le paquebot descend tout seul au ras du sol.

Attention, plus le joueur avance en distance plus le jeu sera difficile !

B – Mise en route du jeu

Le jeu utilise les libraires suivantes :

- SDL2
- SDL2 TTF (rendu des textes)
- SDL2 Image (rendu des images)
- SDL2 Mixer (lecture audio)

Si on se base sur la configuration des machines de Polytech Sorbonne, il n'y a aucune librairie a installer.
Note : Les includes de SDL2 sont sous la forme #include <SDL2/SDL.h> … Pour pouvoir compiler le programme sous les ordinateurs de Polytech Sorbonnes , il est sûrement nécessaire de faire les includes sous la forme : #include <SDL.h> (sans le SDL2/)

Pour compiler le programme, il suffit juste de lancer la commande make dans le répertoire du programme. Une fois compilé, tapez ./gamer pour lancer la jeu. Attention, le jeu commence tout de suite !

II - Étude du programme

A – Vue générale du programme

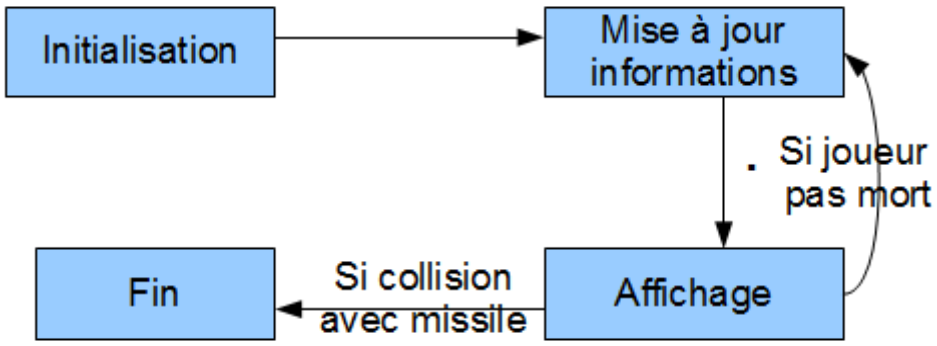


Figure : Vue d'ensemble généralisé du programme

Comme indiqué sur la figure ci dessus, il y a 3 étapes principales + l'étape de fin.
La fonction main() se trouve dans le fichier main.cc

B – Initialisation

Initialisation SDL (graphique) :

Avant de lancer le jeu il faut initialiser les modules de SDL avec ces lignes :

```
//Initialisation des variables principales de la SDL
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO);
TTF_Init();
Mix_OpenAudio(22050, AUDIO_S16SYS, 2, 640);
```

Il faut créer différentes variables nécessaires pour afficher une image à l'écran, notamment:

- SDL_Window : Crée une fenêtre pour contenir un renderer. Elle possède également les propriétés de fenêtre comme la résolution, le titre de la fenêtre,.
- SDL_Renderer : C'est le moteur d'affichage des textures, il gère la position des textures, le moyen d'affichage , définit si on utilise une accélération matérielle ou logicielle etc.

Nous verrons plus tard que pour afficher une image, il faudra également un SDL_Texture, SDL_Surface et SDL_Rect pour chacune de nos images.

Initialisation SDL (timer) :

SDL gère la mesure du temps :

```
while(game.isRunning)
{
    prevTime = currentTime;
    currentTime = SDL_GetTicks(); //Recuperation du temps actuel
    delta = (currentTime - prevTime) / 1000.0f;
```

Selon la charge de l'ordinateur et de l'utilisation des ressources des autres programmes, la durée écoulée entre deux appels d'affichage des textures n'est jamais constante. Afin d'avoir une période de rafraîchissement de l'image constante, on va introduire des variables et utiliser **SDL_GetTicks**
SDL_GetTicks renvoie la durée en millisecondes qui s'est écoulé depuis l'appel à SDL_Init. Ainsi pour obtenir la durée qui s'est écoulé entre deux appels systèmes on fera stocke dans une variable la valeur précédente de SDL_GetTicks puis dans une autre variable la nouvelle valeur de SDL_GetTicks puis on fait la différence.

C – Étude des classes

Classe Game

La classe Game est l'élément central du jeu : Il gère notamment :

- Le gameplay (comportement face aux collisions des objets, la difficulté, la fréquence d'apparition des pièces et missiles etc.)
- Possède les informations sur le joueur comme son score ou la distance parcourue.
- Contient l'ensemble des textures du jeu (hors joueur et fond d'écran) (image + texte).
- La musique de fond

1 – Les conteneurs de textures

La classe Game stocke toutes les classes d'Entités. La classe Game possède donc 3 conteneurs :

Nom du conteneur	Type de conteneur	Raison du choix du type de conteneur
texts	std::map<std::string, Text*>	Si on veut accéder au texte qui affiche le score, il est plus explicite d'y accéder par une clé que par un indice entier.
coins	std::deque<Coin*>	Principe de FIFO (first in, first out). Les premières pièces a apparaître à l'écran seront les premières à partir si ils sortent de l'écran si ils sont pas pris par le joueur
missiles	std::deque<Missile*>	Idem que coins

A noter que pour le conteneur texts, on ajoute les éléments par paire avec std::make_pair(std:string, Text*) , on effet on associe chaque élément à une clé.

2 – Les méthodes de la classe Game

- void Game::eventUpdate(float dt) : Augmente la difficulté et le score à chaque période régulière intervScore.
- void Game::coinsUpdate(SDL_Renderer r, float dt) : Met à jour la position de tous les missiles et génère aléatoirement des rangées de pièces.
- void Game::missilesUpdate(SDL_Renderer r, float dt) : Met à jour la position de toutes les pièces et génère aléatoirement des rangées de missiles.
- void Game::textUpdate(SDL_Renderer r) : Met à jour le texte des distances et score
- void Game::missileDraw() : Affichage de tout les missiles
- void Game::coinsDraw() : Affichage de toutes les pièces
- void Game::textDraw() : Affichage des textes
- void Game::checkColision(Player* p) : Vérifie toutes les collisions des entités avec le joueur et applique l'action à effectuer en cas de colision.
- bool operator==(const SDL_Rect &a, const SDL_Rect &b) : Indique si il y acolision entre deux entités.

Classes Entity (et classes dérivées : Missile, Object, Coin, Player)

Ces classes contiennent toutes les informations sur l'affichage de ces entités (leur comportement est gérée par la classe Game).

La classe Object est dérivée de Entity , elle concerne tout les objets qui interagissent avec le joueur et possède une méthode virtuelle update car le comportement des textures n'est pas la même selon les objets.

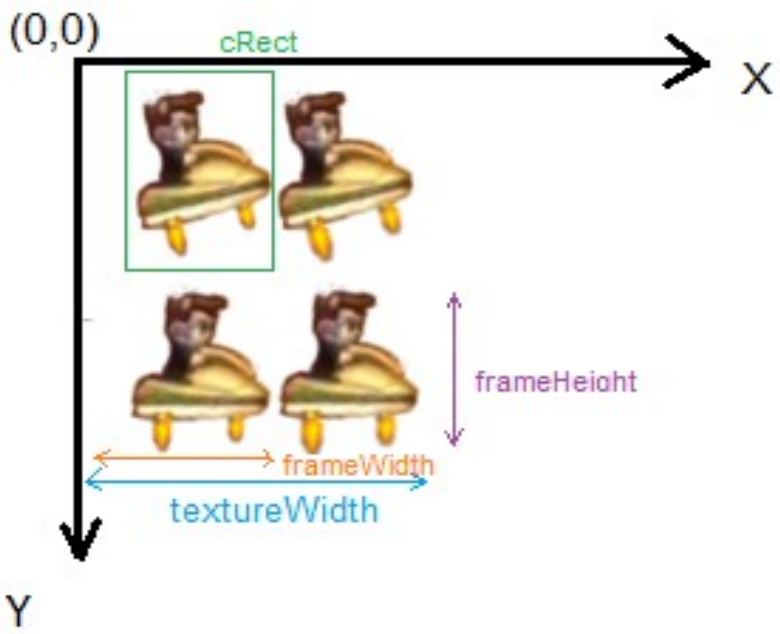
Les classes Coin et Missile dérivent d'Object. A noter que le destructeur de la classe Entity est également virtuelle. Si elle ne l'est pas, les classes dérivés appellent part défaut le destructeur de la classe Entity et cela entraînent des erreurs de compilation. Est ce un bug ?

Chaque classe Entity et dérivée ont une méthode pour faire animer l'entité et le dessiner sur un renderer.

2 – Les membres de la classe Entity

- SDL_Texture *texture: Contient l'image pour y être affichée sur un SDL_Renderer via un SDL_Rect.
- float moveSpeed : Vitesse de déplacement de l'entité
- float frameCounter : intervalle de temps entre deux images dans l'animation de l'entité.
- float frameWidth : Longueur de la texture
- float frameHeight : Hauteur de la texture
- float textureWidth : Longueur Totale de l'image des textures.
- float posX, posY : Coordonnée de la texture sur l'écran
- SDL_Rect posRect : Rectangle d'affichage qui contient une texture, et qui s'affiche sur l'ecran.
- SDL_Rect cRect : Rectangle d'affichage qui sélectionne la texture qui nous intéresse dans l'image des textures.

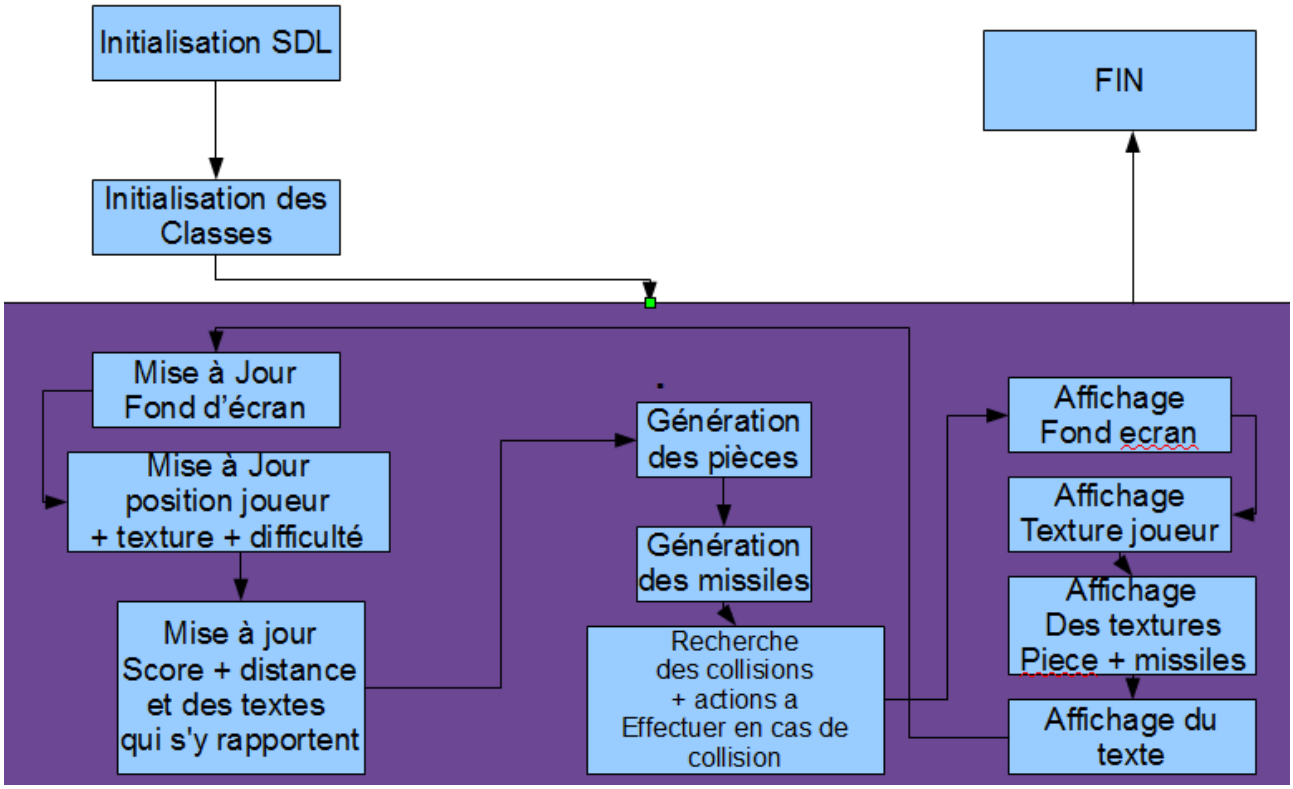
3 – Gestion des textures et des Rect (animation des textures)



L'image des textures regroupe l'ensemble des textures d'une entité
Un rect possède un paramètre x,y (positions) et w,h (largeur et hauteur) . Le cRect choisit la portion de l'image à afficher et posRect l'affiche ce rect sur l'écran.
A noter que dans le constructeur d'Entity, on fournit le nombre de frames en X et Y pour appliquer correctement les subdivisions de l'image. Ainsi a chaque intervalle de temps frameCounter, on déplace le cRect selon l'axe de X pour animer le joueur. De même si le joueur se déplace verticalement on fait déplacer le cRect sur l'axe des Y.

classe Text
Cette classe permet d'afficher les textes sur la fenêtre. Sous SDL, le texte est une image en soi et utilise des SDL_Texture. Nous avons implémenté une surcharge d'opérateur= qui permet de changer le texte de la texture juste via l'affection d'un std::string. Sinon l'affichage se fait de la même manière que les textures des classes d'entity.

C – Exécution du programme.



Tant que le joueur n'est pas touché par un missile, la boucle au sein du rectangle violet continue. Il est très important que l'ordre d'affichage soit respecté au risque d'avoir un fond d'écran qui s'affiche par dessus les pièces par exemple.

III – Conclusion

A – Moments marquants / retour sur expérience

Opérations sur des types différents

Les paramètres de SDL_Rect (w,y,w,h) sont des int. Nous manipulions des floats car les déplacements dx ou dy dépendent de SDL_GetTick et de la difficulté du jeu qui sont en float... Une source important de perte de temps fut l'addition de float avec int qui donnaient un int, et donc troncature les coordonnées et rendaient les déplacement sur +x et +y impossibles car les déplacements 0<dx<1 et 0<dy<1 étaient ignorés. Une erreur élémentaire mais qui nous a pris une journée pour trouver la cause du problème (merci gdb!)

Éléments de conteneur et surcharge d'opérateurs

```
for(std::map<std::string, Text*>::iterator it = texts.begin(); it != texts.end(); it++)//On poarcours les elements des maps avec un iterateur
{
    if((it->first) == "ScoreNumber" )
    {
        *(it->second) = std::to_string(score); //On met a jour les textes avec les valeurs actuelles (Pas de ref. a ce journal de droite svp)
    }
    if((it->first) == "DistanceNumber" )
    {
        std::string dist = std::to_string(difficulty);
        *(it->second)= dist.substr(0, dist.length()-4);
    }
}
```

Afin de pouvoir modifier le texte du score, nous devions accéder aux éléments du conteneur du texte. Ceci n'est pas trop compliqué, sauf quand il fallait l'appliquer à une surcharge d'opérateur.

Ainsi , lorsque l'on faisait un Text* = std::string ; Nous étions confrontés à diverses erreur comme : « can't cast Text to Text* » ou « Class text* lacks a cast » voir un «Segmentation fault » du à SDL_TTF, et dès lors que le compilateur mettait pas d'erreur , la texture du texte ne se mettait pas à jour contrairement à ce qu'affichait un std::cout sur console.

Nous avions bien saisi que l'itérateur it->second pointe sur un pointeur d'une classe. La solution était d'écrire *(it->second) alors que nous avions tenté *it->second … Encore une erreur de syntaxe simple mais qui nous rend à bien confus. Il faut également rappeler que pour un tel projet, il est beaucoup plus pratique de transmettre les classes des textures par référencement car beaucoup plus souple pour changer les infos de textures depuis les méthodes qui se chargent de mettre à jour les informations.

Fierté d'implémentation : Le système d'animation des textures : L'idée de mettre les textures sous forme d'une image unique puis de bouger le cRect selon une période calculé avec SDL_GetTick pour avoir une animation constante est une bonne réussite !

B – Bugs restants et pistes d'améliorations

Le jeu est bien fonctionnel mais nous avons encore quelques bugs :

- A propos de Valgrind : Aucune erreur ou warning n'a été détectée. Mais au bout d'une certaine durée, VALGRIND fait un GENERAL PROTECTION FAULT. Le traçage de la pile montre que cela est dû à libSDL-2.0.0 … Par soucis de matériel, j'utilisais une machine virtuelle qui tourne avec une distribution Lubuntu 17.10 (on ne peut pas installer Linux sur une Surface 2 Pro). Je n'ai pas codé sur Windows par soucis de compatibilités vis a vis des correcteurs et parce que installer la SDL c'est plus simple sur Linux...et puis vive Linux quoi !

- Les collisions sont grossiers, en effet nous avons fait des détections de collisions selon les rect. Or les missiles ou l'image du joueur n'est pas carrée, mais avec des formes irrégulière. Le programme détecte des collisions si les textures sont assez proches. Il aurait fallu détecter les collisions selon la transparence du fond des images ?
- Le sons des pièces est légèrement décalée , car le fichier audio des pièces est décalée.

IV – ANNEXE : Diagramme UML (fichier UML fourni)