Table des matières

1 Comment l'utiliser ?

Le choix de l'outil pour compiler est cmake[1]. Il permet de créer un Makefile adapté à notre machine (donc utilisable sous Windows, Linux et Mac), en plus de rajouter des fonctionnalités tel que la copie de dossier comme Resources et libs si on est sous Windows. Les libraries et includes sous Windows sont fournis dans l'archive. Cependant pour linux, il vous faudra les bibliothèques suivantes :

- SDL2
- SDL2_image
- SDL2_ttf
- SDL2_mixer
- expat

L'utilisation de cmake est définis dans le README pour linux. Sous windows, merci de lire ceci : `https://cmake.org/runningcmake/`. En pratique, il suffira de dire où sont les sources, où es-ce que vous voulez compiler, d'appuyer sur configure et ok. Le Makefile sera ainsi créé et vous pourrez compiler via mingw32-make par exemple.

Le code source est disponible sur `https://github.com/Gaulois94/ET3_Project_C` ou en ligne de commande sous linux : git clone https ://github.com/Gaulois94/ET3_Project_C.git

Les touches du jeu sont définies dans le context Options. Vous pouvez changer les touches en cliquant sur leur valeur (par défaut c'est GAUCHE, DROIT et HAUT pour ce déplacer). Start lancera le jeu et Quit (ou Alt+F4 / croix / killall / etc.) fermera le programme. Le programme aura été testé sous valgrind et possèdera donc théoriquement aucune fuites mémoires.

2 Objectifs

Les objectifs du projet durent principalement d'implémenter les concepts de bases d'un jeu Mario Bros, c'est à dire le fait de traverser une carte créer avec des blocs[2] et des ennemies afin de passer au niveau suivant. En ce sens, nous avons réussi à reproduire ce concept à son stade le plus basique. En effet nous avons pu recréer les systèmes des pièces, des sols, des point de départs (qui n'est certe pas affiché mais existe dans notre format de carte) et des points d'arrivés (le château). De plus, nous avons aussi créé les ennemies les plus basiques appelés Goombas dans le code.

3 Qui fait quoi ?

3.1 Stacy GROMAT

Stacy GROMAT aura créer tout les contexts ainsi que le fonctionnement du joueur. InGame, Start et Options sont donc ses travaux principaux. Elle a de plus coder les classes List et ResourcesManager, qui sont tout simplement des implémentations de std : :list et std : :map (à leur version la plus basique) disponible en C++.

1. `https://cmake.org/`
2. Tiles dans le code

3.2 Mickaël SERENO

L'éditeur de carte étant le sien, Mickaël SERENO aura décompiler la carte, créé les différentes classe Ennemies / Tiles et les classes Drawables / Widgets.

3.3 Ensemble

On ne pouvait pas partir chacun de son coter sans avoir fait ensemble une base solide. Nous avons donc décider ensemble de comment le projet sera construit, comment on gèrera les évènements, comment les classes intéragissent entre elles, etc.

3.4 Internet

Nous n'avons malheureusement pas pu créer nos propres fichiers images, fonts et sons. Tout les fichiers autres que xml présents dans le dossier Resources viennent donc de internet. Les fichiers images viennent toutes de `http://www.mariouniverse.com/`

4 Difficultés / Solutions

4.1 Difficultés

La principale difficulté d'un jeu tel que Mario Bros est de le rendre suffisament modulable pour avoir des perspectives d'améliorations facilement implémentables. De plus, le jeu pouvant vraiment être complet, le format de carte est un choix important du projet.

La SDL nous étant imposés, il fallait la comprendre et surtout l'utiliser le plus simplement possible (dans le sens que le code devait rester logique) pour avoir un projet le plus modulable.

4.2 Solutions

Orienté Objet

Pour la modularité du projet, on a opté pour un approche objet du problème. Bien que le C n'est pas un langage orienté objet, on a pu utilisé les concepts de bases d'un code orienté objet, à savoir :

- Le principe d'héritage
- Le principe de polymorphisme
- Les méthodes et attributs

Pour l'héritage, nous avons tout simplement profiter des caractéristiques des structures en langage C, à savoir que l'on peut caster et avoir accès aux N premiers bits d'une structure comme si celle ci était un autre type de données.

Pour le polymorphisme, les pointeurs sur fonctions étaient ce qui semblaient le plus appropriées. On a juste à changer ce pointeur (qui fait partie des attributs de la structure) vers la bonne méthode et tout fonctionnera.

Enfin, on accèdes aux attributs de la classe mère via des cast (principe qui consiste à changer dynamiquement le type d'une variable) et donc d'y accéder. En effet pour le compilateur, les attributs ne sont tout simplement que des décalages de mémoires depuis le début de son adresse et une taille associé à un nom d'attributs. Caster donc une variable nous permet donc d'utiliser ces noms et donc de changement / récupérer les attributs de la classe mère. Pour les méthodes, on a choisie la convention suivante : ¡type¿Classe_nomFonction(¡type¿ nomAttributs) ; qui nous semblait le plus lisible.

Moteur de jeu

Avec le concept d'objet utilisable, nous avons pu développer un petit mais fonctionnel moteur de jeu. Les animations, textes, sprites, Widgets sont utilisables dans l'ensemble, et ne dépende pas du jeu lui même.

On a de plus utilisé le concept de Context afin d'avoir des parties du code bien définie. Un Context est tout simplement un morceau du jeu qui se suffit à lui même. Le menu Start, le menu d'Options et le jeu en lui même (InGame) sont des Contexts qui ne dépendent de personnes pour subsister. Les variables qu'ils échangent sont enregistré dans globalVar.c / globalVar.h

Pour décompiler la carte, nous avons utiliser expat [3] qui est un parser évènementiel. À chaque fois qu'il rencontre une balise ouvrantes (fermantes), il appelera la fonction correspondante définie par XML_SetElementHandler en donnant en paramètre une donnée qu'on aura choisi (ici se sera notre structure Map), les attributs de la balise et son nom. Le format de la carte est donné en commentaire dans include/Map.h.

Nous en avons profité pour créer un parser CSV basiques. En effet, les rangés de blocs sont écrites en CSV, et il nous fallait donc écrire un parser.

Les évènements

Pour les évènements, nous n'avons pas voulu qu'une seul entité gère tout les évènements. Nous donnons donc successivement les évènements aux objets qui peuvent potentiellement récupérer à la voler cet évènement et laisser cette entité gérer son évènement toute seul. Comme les évènements au clavier ne sont pas continues, nous utilisons des variables en plus pour connaître le dernier état d'un évènement (par exemple stillDown dans Player.h qui permet de savoir si on est encore appuyé sur la touche).

3. `http://expat.sourceforge.net/`