M1103 : Projet de Programmation

Monster

Hourquet Jordan – Bascouzaraix Julien
S1B

2016 - 2017

Enseignant Responsable : Nicholas Journet

Table des matières

1. Introduction

1.1. Contexte

Ce projet vient clôturer le premier semestre du DUT. Il compte dans la note en Algorithmie-Programmation pour valider l'Unité d'Enseignement en Informatique.

C'est le second projet depuis le début de l'année dans cette matière, il a débuté à la suite directe du précédent qui avait pour consigne de coder en C++ une copie du jeu « Pong ».
À compter de la semaine du 14 novembre, nous avons eu environ 1 mois pour effectuer le projet jusqu'à la date limite du 15/12/2016.

1.2. Objectifs

Les objectifs pour ce projet étaient principalement d'évaluer et de valider la mise en application des notions vues depuis le début de l'année en algorithmie-programmation. Contrairement au précédent projet qui consistait en une succession d'ateliers ayant pour but de nous initier aux projets en général et à quelques notions primordiales (librairies « SDL »,...), le projet « Monster » s'effectue dans une quasi-totale autonomie.

Il était également question d'initier à la programmation en groupe, afin de préparer à ce qui sera un aspect primordial dans le secteur professionnel de la programmation et de développer des qualités comme la communication, l'esprit d'équipe, l'organisation ou le « leadership ».

1.3. Outils utilisés

Pour le projet, l'intégralité du code est écrite en C++. Sur le site du département informatique, l'IDE par défaut pour du codage en C++ est QTCreator. Nous utilisons cet IDE depuis le début de l'année en cours de TD et TP ainsi que pour le projet précédent, nous avons donc poursuivi sur ce logiciel.

Comme librairies notables utilisées au cours du projet, hormis les bibliothèques classique comme « iostream », nous avons utilisé la librairie « array » nous permettant de créer des tableaux ainsi que les bibliothèques permettant de manipuler les fonctions de la SDL afin de gérer et d'utiliser des images.

2. Implémentation

2.1. Architecture générale

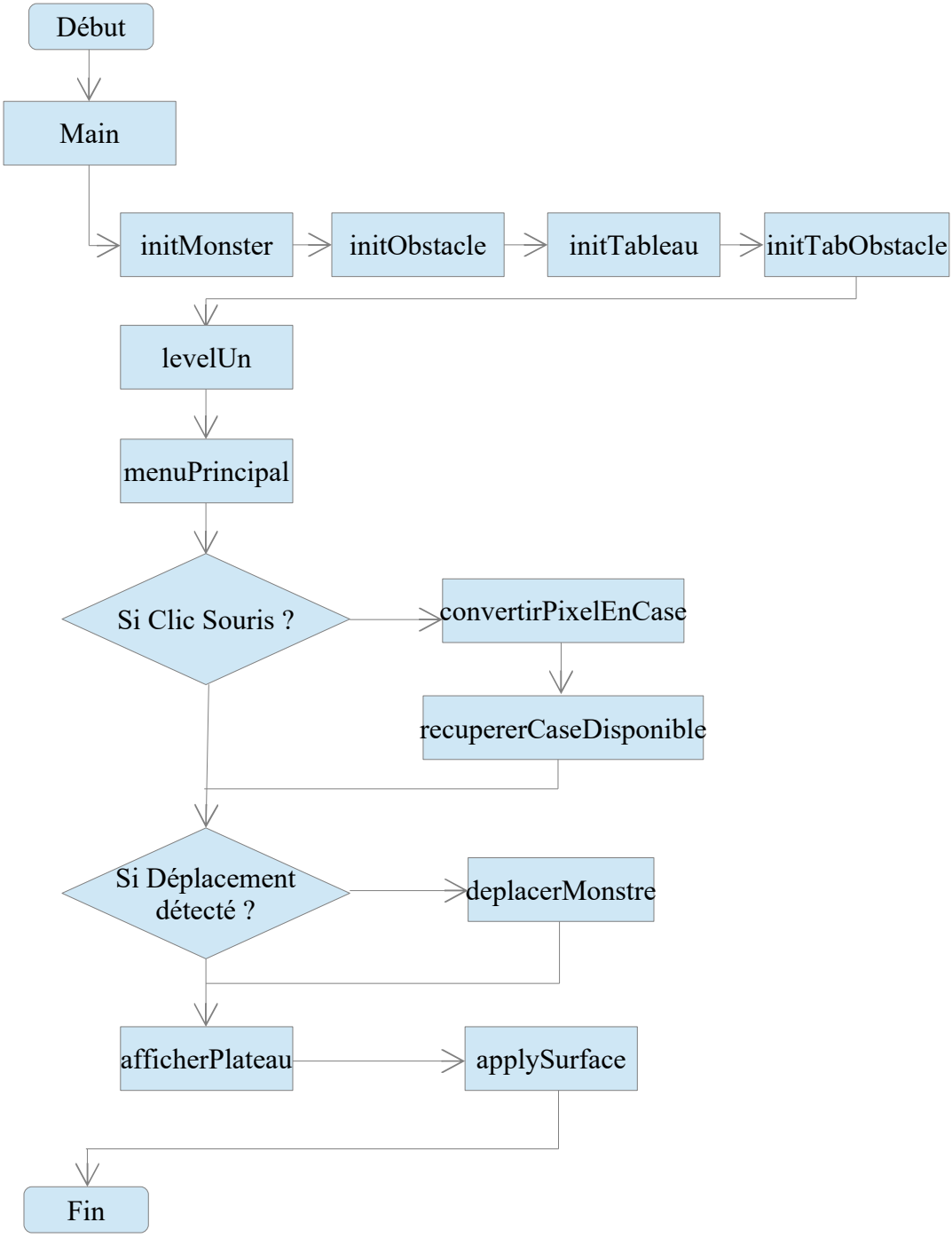
Notre projet se compose en tout de 9 fichiers. Les fichiers « .h » contiennent les structures, les constantes, les tableaux et permettent de faire le lien avec les autres fichiers au niveau des appels de fonctions. C'est dans les fichiers « .cc » que se trouve le code des fonctions en lui-même et donc les instructions à effectuer lors de l'appel de l'une d'entre elles.
Le fichier *main.cc* comporte la majorité des appels de fonctions, les fonctions importantes ainsi que la boucle de jeu.
Nous avons ensuite un premier couple de fichiers *SDL.h* et *SDL.cc* qui servent à faciliter l'utilisation de la bibliothèque SDL dans les différents fichiers du projet.
Le couple *plateau.h* et *plateau.cc* gère l'affichage en jeu, le calcul des cases et des coordonnées et leur correspondance avec l'affichage des éléments. Dans *plateau.h* sont déclarées les structures contenant les coordonnées en pixels et le numéro des cases du tableau ; on y déclare également les différents types d'éléments (monstres ou obstacles) correspondant chacun à une valeur numérique ainsi que les tableaux utilisés et enfin les valeurs constantes. C'est dans le *plateau.cc* qu'est géré l'affichage en jeu et les différentes fonctions permettant de calculer les cases sur lesquelles sont situés des éléments et de convertir des coordonnées de cases en pixels.
Dans le couple *monster.h* et *monster.cc*, on déclare les structures pour les monstres et les obstacles dans le premier et dans le second, on initialise les différents sprites et on gère le déplacement des monstres en fonction de la direction détectée par le programme et donnée par le déplacement de la souris.
Le dernier couple *affichage.h* et *affichage.cc* contient uniquement le menu principal du jeu (écran de démarrage du monster).

2.2. Structure de données

Dans le fichier *monster.h* se trouve la structure permettant de déclarer les monstres et celle permettant de déclarer les obstacles.
La structure Monstre contient des coordonnées x et y, correspondant au placement des monstres sur l'axe des abscisses et l'axe des ordonnées en pixels. Elle contient également des rectangles qui délimitent l'affichage des différents monstres (monstre rouge et monstre endormi) par rapport au fichier *sprite.bmp* qui contient tous les sprites des éléments du jeu.
La structure Obstacle contient uniquement les rectangles délimitant l'affichage des sprites de la glace, des livres et des flèches (ces dernières ne sont cependant pas utilisées dans le code).

Le fichier *plateau.h* contient deux autres structures. Dans la structure caseTableau, nous avons déclaré deux variables qui vont correspondre à des numéro de cases qui serviront de repères dans les tableaux d'entiers à propos de la position de certains éléments.
La structure corresCasePixel contient aussi uniquement des variables d'entiers qui serviront à convertir des coordonnées de case dans un tableau en coordonnées en pixels sur l'affichage.

2.3. Organigramme d'appel des fonctions


Début
Main
initMonster
initObstacle
initTableau
initTabObstacle
levelUn
menuPrincipal
Si Clic Souris ?
convertirPixelEnCase
recupererCaseDisponible
Si Déplacement détecté ?
deplacerMonstre
afficherPlateau
applySurface
Fin

3. Aspects Algorithmiques

3.1. Algorithme 1

Nous avons choisi de vous présenter en premier la fonction recupererCaseDisponible.

```
210  /****************** Nom de la fonction *********************
211  * recupererCaseDisponible                                  *
212  ******************** Auteur , Dates ************************
213  * Bascouzaraix Julien Version 6                            *
214  ********************* Description ***************************
215  * Algorithme qui permet de trouver la case                 *
216    ou le monstre doit s'arrÃªter dans son Ã©lan               *
217  ********************** EntrÃ©es ****************************
218  * Le tableau de monstre et d'obstacle, la case Initiale    *
219    La structure corresCasePixel et un entier : reset        *
220  ********************** Sorties ****************************
221  * Toutes les entrÃ©es                                       *
222  ************************************************************/
223  void recupererCaseDisponible(TGrille & tableau , TGrille2 & obstacles, caseTableau & caseInitiale, corresCasePixel & pos, int & reset)
224  {
225      /** Avant de lire cette fonction, il faut savoir que dans notre cas, les colonnes sont les i et
226       * les lignes sont les j, la confusion peut-Être faite **/
227
228      //Initialise la dernière case disponible
229      reset = 0;
230      int lEnd=caseInitiale.j;
231      int cEnd=caseInitiale.i;
232
233      //bas
234      if(caseInitiale.i == pos.i && caseInitiale.j < pos.j)
235      {
236          while(tableau[cEnd][lEnd+1] == INVISIBLE && obstacles[cEnd][lEnd+1] == NUL && lEnd <= TAILLE_Y)
237              lEnd++;
238          if(lEnd >= 8)
239          {
240
241              reset = -1;
242          }
243      }
244
245      //haut
246      else if(caseInitiale.i == pos.i && caseInitiale.j > pos.j)
247      {
248
249          while(tableau[cEnd][lEnd-1] == INVISIBLE && obstacles[cEnd][lEnd-1] == NUL && lEnd >= 0 )
250              lEnd--;
```

Un des algorithmes les plus importants de notre projet, cette fonction réalise une incrémentation d'une des quatre directions (soit de i (les colonnes) soit j (les lignes)) jusqu'à rencontrer un obstacle et ainsi nous donner les nouvelles coordonnées de la case destination. Pour cela, nous avons en entrée, Deux tableaux(le tableau de monstre et d'obstacle), les coordonnées de la case initiale et de la case destination, une structure corresCasePixel et enfin un entier.
L'algorithme fonctionne de la manière suivante, on regarde si deux j sont différents ou deux i sont différent, suivant la valeur de i ou j de caseInitiale et pos, une fois cela effectué, nous savons ou notre monstre va se dirigé en effectuant l'incrémentation jusqu'à l'obstacle.
De plus, nous avons rajouté que si l'incrémentation se fait jusqu'à des cases d'extrémité du tableau (ce qui est impossible à cause des règles du jeu : si un monstre sort du tableau, le level et réinitialiser) l'entier en entrée prend la valeur -1 et dans le *main.cc* si cette entier vaut -1, le level et réinitialiser.

3.2. Algorithme 2

La deuxième fonction que allons décrire est la fonction *deplacerMonstre*.

```
 * deplacerMonstre                                          *
 ******************** Auteur , Dates ************************
 * Basconzaraix Julien,  Version 5                          *
 ******************** Description ***************************
 * DÃ©place le monstre sur le tableau et gÃ¨re les diffÃ©rentes *
 *          condition du jeu                                *
 ********************** EntrÃ©es ****************************
 * une structure Monstre, les coordonnÃ©es de la case        *
 *  Initiale et destination et le tableau de monstre et     *
 *  d'obstacle                                              *
 ********************** Sorties ****************************
 * La structure Monstre et le tableau de monstre            *
 ************************************************************/
void deplacerMonstre (Monstre &m, caseTableau caseInitiale, caseTableau caseDestination, TGrille & tableau, TGrille2 & tableau2)
{


    const int velocity_x = 3, velocity_y = 3;


    // bas
    if(caseInitiale.i == caseDestination.i && caseInitiale.j < caseDestination.j)
    {

        if(tableau[caseDestination.i][caseDestination.j] == tableau[caseInitiale.i][caseInitiale.j])
        {

        }
        else
        {
            m.y += velocity_y;
            SDL_Delay(100);
        tableau[caseDestination.i][caseDestination.j] = MON_MONSTRE;
        tableau[caseInitiale.i][caseInitiale.j] = INVISIBLE;
        }
        if(tableau[caseDestination.i][caseDestination.j+1] == ENDORMI)
            tableau[caseDestination.i][caseDestination.j+1] = MON_MONSTRE;
        if(tableau2[caseDestination.i][caseDestination.j+1] == GLACE)
            tableau2[caseDestination.i][caseDestination.j+1] = NUL;


```

Suite logique de la fonction *recupererCaseDisponible,* Cette fonction réalise le déplacement du monstre vers la case destination. Nous avons en entrée, la structure du monstre, la case initiale, la case destination ainsi que les deux tableaux.
L'algorithme fonctionne de la manière suivante, on reprend les mêmes conditions que *recupererCaseDisponible* pour regarder où le monstre se dirige, une fois ces conditions effectuées, on remplace la case initiale par du vide et la case destination par le monstre. S'en suit deux conditions : si l'obstacle qui l'a arrêté est un monstre endormi, il devient un monstre jouable, si c'est un bloc de glace, il se détruit et la case devient vide.

4. Difficultés rencontrées

Nous avons au cours du notre projet de nombreux problèmes, liés pour la plupart à un mauvais choix technique au commencement du projet qui nous a contraint de revoir la majorité de notre code au milieu du projet.

Un de nos premiers problèmes s'est situé au niveau du choix des structures de départ.
Il faut savoir que nous avons choisi de ne pas utiliser les exemples de code qui étaient donnés dans les ateliers concernant le projet sur la plateforme Moodle, notamment pour les différentes structures de base à utiliser, choix discutable étant donner la suite des événements.
Nous avons alors choisi de créer une structure unique qui devait uniquement servir à déclarer tous les sprites que nous allions utiliser. Nous nous sommes vite rendu compte que ce n'était pas pratique et nous sommes allés vers deux structures (une structure pour les monstres avec des coordonnées et des sprites ainsi qu'une structure d'obstacles avec uniquement des sprites puisque les obstacles ne bougent pas).
Cependant nous avions déjà commencé à coder certaines fonctions pour l'affichage du jeu par exemple, et nous n'avions pas non plus crée de tableau d'entiers.
Le problème à ce moment est que nous étions partis sur une mauvaise stratégie technique, nous cherchions des solutions pour attribuer des coordonnées à un sprite sans passer par un tableau. Cette stratégie nous a amené vers de nombreuses fausses pistes.
On a ensuite du adapter notre code au fur et à mesure que nous rattrapions des bonnes pistes, tout d'abord en ajoutant un tableau d'entiers pour aider au placement des différents éléments en fonction de la grille du jeu puis en divisant les structures.
Mais l'accumulation de ces fausses pistes qui nous fait perdre énormément de temps et commencé à jouer sur les nerfs et sur la motivation, nous n'arrivions pas à identifier les problèmes, cela bloquait la continuité de notre projet et notre organisation a alors été quelque peu chamboulée à la suite des événements.
Il nous a fallu l'intervention des professeurs pendant les cours de projet ainsi que des tuteurs de S3 en tutorat le mercredi après-midi pour nous remettre sur la bonne voie. Malheuresement ce fût relativement tardif.
À ce stade, l'idéal aurait été de reprendre entièrement notre code mais le temps était déjà compté.

5. Conclusion

Cette section devra faire écho à l'introduction : avez-vous fait ce que l'on vous a demandé de faire ? Avez-vous respecté les délais ? etc
1 à 2 pages maximum

Au final, malgré de nombreuses difficultés qui paraissaient relativement insurmontables à certains moments, le résultat final est fonctionnel bien que loin d'être aussi poussé que celui d'autres groupes. Notre code est relativement restreint, nous n'avons pas inclus l'usage des différentes flèches (bien que nous ayons défini leur rectangle d'affichage) et nous n'avons fait qu'un seul niveau. Cependant, aux vues des difficultés auxquelles nous avons du faire face, le code final satisfait tout de même plusieurs des critères d'évaluation.
En ce qui concerne les délais, nous avons respecté ces derniers même si du coup notre projet n'est pas aussi abouti que ce que l'on aurait voulu. Le projet est rendu le 15/12/2016, date limite pour la restitution du projet sur la plateforme Moodle.