Rapport Final : Projets Technologiques

Théo Dupont, Elliot Renel

21 avril 2019

Table des matières

1 Introduction

Bonjour, nous vous présentons ici le rapport final concernant l'UE Projets Technologiques et plus particulièrement l'utilisation de divers outils informatiques tels que git, CMake ou encore LaTeX, mais également l'implémentation du jeu Net, accessible en ligne ici : www.chiark.greenend.org.uk/~sgtatham/puzzles/js/net.html. Jeu inventé par David Millar, dont l'accès est fourni par Simon Tatham.

Le principe de ce jeu est de relier toutes les pièces entre elles en un seul bloc, en faisant tourner celles-ci. Nous avons tout d'abord fait une version de "base" texte pour finalement étendre le jeu a plusieurs possibilités de jeu différentes et avoir une version graphique Nous avons également réalisé un solveur, permettant de résoudre automatiquement le jeu que l'on choisit (si il a une solution évidemment). Finalement, on a réalisé un portage Android pour permettre au jeu de fonctionner sur smartphone de la même manière que sur ordinateur.

Nous allons donc détailler toutes ces étapes ainsi que d'autres dans les sections qui suivent.

2 Version texte du jeu

On nous a tout d'abord donné une bibliothèque game fonctionnelle pour réaliser la version texte du jeu, c'est à dire faire en sorte d'afficher la grille de jeu et d'utiliser les différentes fonctions de la bibliothèque pour en faire un jeu fonctionnel. On avait pour cela une grille "modèle" pour nous montrer le résultat attendu. L'implémentation consistait ainsi à pouvoir afficher et jouer à un jeu donner en exemple dans le fichier *game.h*. L'algorithmique de l'affichage n'était pas bien compliqué, et consistait à réaliser 2 boucles *for* pour parcourir la grille du game de haut en bas et de gauche à droite. Nous avons eu quelque bugs pour l'affichage de certains caractères comme les Tee et les corner car ils étaient traités comme 2 caractères au lieu d'un, et ont donc été initiés comme *string* au lieu de *char*. La fonction main était la seconde grande partie de cette implémentation, car c'est elle qui gère le bon déroulement du jeu. Une simple boucle *while* avec *is_game_over()* et un *scanf* pour les coordonnées était suffisant, avec bien sûr les tests pour savoir si les coordonnées étaient valides.

Pour le résultat du travail, on avait un jeu fonctionnel, avec la grille qui s'affiche correctement et on pouvait bien rentrer les coordonnées pour

modifier la direction des pièces. Un bug présent dès ce moment, mais que l'on apprendra que bien plus tard, lors de l'évaluation par les pairs, est lors de l'entrée des coordonnées. En effet, si le texte entré n'était pas un nombre par exemple, ou un nombre trop gros pour le jeu, le jeu commençait à mal fonctionner voire plus du tout. On a donc corrigé ce bug après l'évaluation par les pairs avec la proposition de Monsieur Blin, en utilisant *getc()* à la place de *scanf()*.

3 Implémentation de la bibliothèque game

L'implémentation de la bibliothèque game consistait à implémenter les différentes fonctions nécessaires au bon déroulement du programme.

On a reçu pour cela l'outil git, que l'on avait depuis le début du projet, mais que l'on a vraiment commencé à exploiter à partir d'ici. En effet cela allait nous permettre de diviser le travail pour les différentes fonctions à réaliser, et cela de manière automatique et simple. On a d'ailleurs utilisé sans arrêt les fonctionnalités de git tout au long du projet pour progresser et tester nos implémentations. On a par exemple créé des branches pour faire des expérimentations sur les nouvelles fonctionnalités demandées, ou pour faire une partie de travail à part comme l'implémentation graphique par exemple.

Donc, pour revenir à l'implémentation de la bibliothèque game, on a implémenté toutes les fonctions du fichier *game.c*. On s'est divisé le travail, à l'époque on était trois dans le groupe. Chacun à pris des fonctions et on a fait les fonctions de notre côté, car chaque fonction était déjà définie par les professeurs, il ne fallait donc qu'interpréter chaque fonction et l'implémenter en conséquence.

On a ainsi implémenté notre propre structure pour le jeu, que l'on a un peu amélioré tout le long du projet. On a aussi implémenté une structure pour permettre de modéliser les pièces même du jeu (que l'on a appelé *node*). La plupart des fonctions n'ont pas vraiment posé de problèmes car beaucoup n'était que des fonctions *"get"*.

La plus grosse difficultée a été dans l'implémentation de *is_ game_ over()*. En effet, de par sa nature algorithmique, elle a nous a demandé du temps lors de son écriture, ainsi que plusieurs fonctions annexes. Pour faire simple, *is_ game_ over()* parcourait le jeu à partir de la case (0,0) et "voyageait" de case en case récursivement jusqu'à atteindre une leaf. Si la case vue par la

fonction a déjà été parcourue, ou si elle est connectée à rien, ou si on ne parcourt pas toutes les cases du jeu, alors *is_game_over()* retourne false, et true sinon.

Un bug très bizarre que l'on a eu lors de cette implémentation a été le fait que la fonction *shuffle* exécutait le même mélange si relancé 2 fois d'affilée. Pour le résoudre il a fallu initier le *seed* avec les microsecondes de l'horloge pour avoir un résultat différent à chaque exécution.

L'une des pistes d'amélioration que l'on avait était d'utiliser la fonction *new_game_empty()* dans *new_game()* afin d'éviter de faire de la redondance de code.

Dans cette partie, on a eu un amphithéâtre nous présentant ce qu'est une IDE mais également le logiciel Visual Studio Code qui fut d'une grande aide pour le développement du projet, avec un nombre de fonctionnalités énormissime et une personnalisation gigantesque. De plus, l'utilisation de git avec VSCode est bien simplifiée par rapport à son utilisation avec le terminal. Un autre logiciel que nous avons utilisé tout au long de l'année est gitk, qui permet d'avoir une meilleure vision de ce qu'est git et des différentes branches créées.

4 Extension de la bibliothèque (V2)

Après l'implémentation de la bibliothèque, on a reçu de nouvelles fonctionnalités à implémenter par dessus la bibliothèque que l'on a créée. La ligne directive qui nous a été donnée et que l'on a suivie est la définition de variables et le remplacement des "nombres magiques" par ces variables. Ainsi on aurait un jeu plus flexible, pouvant être facilement modifiable si on voulait modifier la taille du jeu, ou ajouter par exemple une direction, ou une pièce, ce qui était le cas pour la pièce en croix. Pour cette V2, on a dû également ajouter un "mode de jeu" torique. La difficulté était de tout "normaliser" pour s'adapter à un nombre de pièces et de directions variable, ainsi que la taille du jeu et le mode de jeu torique ou non.

Une grande évolution que l'on a eu lors de l'implémentation de la V2 a été l'utilisation d'un CMake à la place d'un MakeFile. Ce changement, d'abord mal accueilli puis vu comme indispensable, a permis de drastiquement simplifier le management des dépendances et la création des exécutables.

L'implémentation du mode de jeu torique nous a permis de révéler un problème de conception dans *is_game_over()*. En effet, pendant son "voyage",

la fonction gardait en mémoire la direction d'origine (afin de ne pas retourner sur ses pas), et la position (0,0) avait une direction d'origine à Sud. Cela ne posait pas trop de problème pour un jeu à bord, mais cela en posait un gros pour un jeu torique car la direction Sud était une direction valide à la position (0,0). Cela faisait que pour certains jeux finis, *is_game_over()* ne parcourait pas toute une partie du jeu (et disait donc que le jeu n'était pas fini). On a ainsi changé cette initialisation afin qu'il puisse aller dans n'importe quel direction à la position (0,0).

5 Tests

Après avoir écrit notre implémentation de jeu en version texte, on nous a demandé de réaliser des tests pour chaque fonction de la bibliothèque game. On s'est alors découpé de nouveau le travail et on a fait chacun un fichier de test avec les fonctions que l'on traitait. On a ensuite fait en sorte d'utiliser les tests pour vérifier nos fonctions. Pour cette partie, il n'y a pas eu de soucis en particulier. Un axe d'amélioration serait de mieux tester nos fonctions, et de continuer à réaliser des tests pour le reste du projet, voire pour certaines fonctions annexes, comme celles de *is_game_over()*.

Après le passage vers la V2 et l'utilisation du CMake, l'ajout des tests au CMake a été assez ardu et nous a demandé beaucoup de recherche afin de comprendre son fonctionnement, mais une fois fait cela a grandement simplifié la visualisation des résultats des tests.

C'est à partir de l'implémentation et l'utilisation des tests sur notre bibliothèque que l'on a commencé à sérieusement utiliser des logiciels de débogage tel que gdb ou des logiciels vérifiant les fuites mémoire comme valgrind. Le débogage nous a ensuite servi à toutes les étapes suivant celles-ci, en passant des implémentations des bibliothèque jusqu'au portage Android, souvent pour trouver des erreurs assez stupides.... On a particulièrement eu des problèmes concernant les fuites mémoires. En effet, à plusieurs reprises, on utilisait des objets *cgame* sans les supprimer à cause du fait que c'était des constantes. Cela avait donc généré plusieurs fuites de mémoires que l'on a résolu en changeant les objets *cgame* en *game*, et ainsi les supprimer plus facilement. Cependant , nous n'avons pas utilisé des outils de coverage comme gcov.

6 Solveur

Le but de la partie sur le solveur était de créer un fichier *net_ solve.c* qui permettait de résoudre un jeu passé en paramètre. Il devait pour cela pouvoir faire 3 choses :

— trouver une solution pour le jeu
— trouver toutes les solutions du jeu
— trouver le nombre de solutions du jeu

Pour l'algorithme, nous avons décidé d'implémenter une recherche *"brut force amélioré"*, c'est à dire une recherche en testant toute les possibilités de jeu, mais en abandonnant directement les configurations voué à donner un jeu non fini. En pseudo-code cela donne :

```
function find_solution(game,x,y) {
    // Les prochaines valeurs de positions dans le game
    x2 = next x;
    y2 = next y ;
    end = y2==game_height; // Booléen détectant si on a atteint la fin du jeu
    for i in range(NB_DIR){
         rotate_piece_one(game,x,y);
         if doomed(game,x,y){
              // Revient dans le for pour incrémenter i
              continue;
         }
         if end && is_game_over {
              save_game;
              return true;
              // Ou autre action selon ce que l'utilisateur demande
         }
         elif not end {
              status = find_solution(game,x2,y2);
              if status
                   return status ;
         }
    }
    return status ;
```

```
}
```

A noter que la fonction *doomed(game,x,y)* vérifie si l'orientation de la pièce à la position (x,y) garantit que le jeu n'aura pas de solution. Cela permet ainsi d'éliminer un bon nombre de possibilités condamnées. La fonction en elle même consiste à une suite de switch qui permettent de dire dans quelle de figure l'on est selon 9 possibilités de position x et y (divisé en 3 possibilité pour x et 3 pour y) :

—

$$x = 0 | y = 0 \tag{1}$$

—

$$0 < x < w - 1 | 0 < y < h - 1 \tag{2}$$

—

$$x = w - 1 | y = h - 1 \tag{3}$$

Ainsi, dans chaque cas (3x3 cas) on vérifie certaines directions afin de savoir si des pièces déjà *"fixées"* sont en accord avec la pièce actuelle (par exemple CW SN serait invalide car le corner en direction west ne sera jamais connecté au segment en orientation nord).

En terme de bug nous en avions quelques un pour les grilles non carrés dû à une inversion des paramètres height et width, très vite corrigé.

7 Jeu version graphique

Pour l'implémentation en mode graphique, on utilise la librairie SDL2. On a dû alors implémenter les différentes parties *init()* , *render()*, *process()* et *clean()* nécessaires pour avoir un jeu en mode graphique avec la librairie.

- *init()* permet d'initialiser les différents paramètres tels que les textures ou les positions de départ
- *render()* permet l'affichage des textures et du texte que l'on veut afficher
- *process()* réalise "l'évolution" des paramètres en fonction des actions du client
- *clean()* supprime les allocations pour les textures et éventuels éléments du jeu.

Après avoir compris comment utiliser chaque partie, on a traité chacune des parties indépendamment, en partant du modèle donné lors du TP sur SDL2, on a modifié le fichier de sorte à avoir ce qui suit :

- *init()* crée l'environnement, ajoute le game à l'environnement selon les paramètres donnés lorsqu'on lance le programme, initialise les positions de la grille par rapport à la fenêtre et initialise toutes les textures et texte que l'on va utiliser. Pour notre implémentation graphique, nos pièces sont des images que l'on va disposer sur la grille et faire tourner en fonction de la direction de la pièce.
- *render()* fait une double boucle pour afficher chaque pièce en partant des positions initialisées. Il réalise aussi l'affichage du message de fin quand le jeu est fini.
- *process()* vérifie sur quelle case de la grille on tape, et modifie donc le jeu en conséquence.
- *clean()* détruit tout simplement toutes les textures initialisées au départ, ainsi que le game et finalement l'environnement utilisé.

Ces quatre fonctions sont alors utilisées dans un autre fichier donné en tant que modèle, que l'on lance pour avoir l'affichage complet de l'application. On intégrera plus tard trois boutons, *New Game*, *Solve* et *Reset* Le premier crée un nouveau jeu aléatoire avec les mêmes paramètres que le précédent, le second résout le jeu et le troisième redémarre le jeu à zéro.

Le principal problème de cette partie était l'affichage correct de notre grille dans la fenêtre. On a utilisé la fonction *SDL_RenderCopyEx()* qui nous a permis de réaliser les rotations des pièces simplement mais les explications liées à celle-ci étaient assez peu détaillées, on a du donc expérimenter.

Plusieurs bugs se sont manifestés liés à la taille des pièces et du jeu évolué avec la taille de la fenêtre. Par exemple, si le jeu n'était pas carré, il dépassait sur les bords, il a donc fallu prendre en compte le format de la fenêtre par rapport à celui du jeu pour pouvoir agir en conséquence. Un autre décalait les pièces légèrement, et donner un aspect visuel très vilain. Ce dernier a aussi été corrigé, il était causé par un arrondie de *double* vers *int* incorrect.

Pour les améliorations on a pensé à des boutons, que l'on a fini par implémenter, et faire une modification du bouton new Game ou l'on pourrait rentrer de nouveaux paramètres pour créer un game différent du précédent (par sa hauteur ou si il est torique par exemple), mais on n'a pas encore implémenté cette fonctionnalité.

8 Portage Android

Ici on a simplement restructuré le projet de manière à ce qu'il convient au portage et on a modifié la version graphique de sorte à placer une partie Android pour que SDL utilise les bonnes fonctions lorsqu'on lance depuis un Android. Ici, on a encore pris pour modèle celui donné en TP, pour la restructuration notamment. Après avoir fait les modifications adéquates pour que le programme puisse fonctionner normalement sur un smartphone, on a fait du débogage car l'application se fermait dès qu'on la lançait. Après avoir trouvé quelques erreurs (format des images mauvais, mauvaise utilisation des fonctions...), l'application se lance correctement, avec l'affichage convenu. Quand on lance l'application, un jeu 5x5 non torique aléatoire se lance, et l'on peut jouer directement.

Un bug que nous avons trouvé est lorsque l'on veut tourner l'écran du smartphone virtuel, l'application n'aime pas du tout et je suis obligé de relancer l'application.

Une piste d'amélioration pourrait être de corriger ce bug, et de faire un menu pour choisir les options de notre jeu (taille, torique ou non) avant de lancer le jeu.

9 Conclusion et Commentaire

9.1 Théo Dupont

Cette UE est très intéressante pour son aspect plus "Réel" que le reste des matières, mais cela à tendance à nous perdre. Des explications de 5 min au tableau sur le makefile, cmake ou la portabilité en plus de la lecture des diapos ne seraient à mon avis pas de trop et permettraient de mieux comprendre ces notions. Peut être également faire des sorte de différences entre ce que l'on faisait en L1 et ce que l'on va faire. (Exemple comment on passe de gcc ..... à Makefile à Cmake) Pour l'utilisation des objets technologiques, même si ils sont géniaux, je dirais peut être de les faire découvrir durant le S2 si possible car pour des élèves n'étant pas de base baignés dans l'informatique comme moi, découvrir VSCode ou git ne m'aurait probablement pas fait de mal et j'aurais peut être pu plus facilement assimiler les notions que l'on applique dans cette UE.

9.2 Elliot Renel

Pour moi, cette UE a été grandement bénéfique pour plusieurs raisons grâce aux différent outils utilisé comme Git, CMake, gdb, valgrind, VS code, SDL2, LaTeX, etc. Tous ces outils m'ont permis de sentir un peu plus à quoi ressemblait un véritable projet de programmation, et me permettront d'être beaucoup plus productifs dans de futures projet personnel ou professionnel.

Elle m'a aussi permis de m'améliorer grandement en C, et de travailler en groupe pour un projet. L'aspect application poussé de l'UE m'a aussi beaucoup plus car elle permettait d'assimiler la connaissance des outils utilisé tout en ayant la satisfaction d'avoir créé quelque chose de réel.

Merci d'avoir lu notre rapport, et merci pour votre soutien durant toute l'année.