Rapport du projet Mario Sokoban

Pierre AYOUB

30 décembre 2016

1 Fonctionnement du Sokoban

Cette partie sera consacrée au fonctionnement interne du Sokoban et à la manière dont j'ai résolu certains problèmes, afin de répondre au cahier des charges. Le projet est organisé comme suit : l'exécutable sera produit dans le dossier racine (à côté du Makefile), les images sont stockées dans le dossier « sprites », les sources dans le dossier « src » et les headers dans le dossier « inc ». Les headers contiennent les définitions des structures, des énumérations et des constantes de préprocesseur, ainsi que les déclarations des fonctions du module et la description du rôle de chaque fonction (documentation). Les sources contiennent le code source et les commentaires nécessaires à la compréhension du fonctionnement interne de la fonction, si besoin est.

Ma philosophie de programmation lors de ce projet était de faire un développement organisé et rigoureux, à l'aide d'outils tel que GDB, Valgrind et Git. Le Sokoban est programmé pour gérer tous les cas d'erreur qu'il pourrait rencontrer, mais aussi de manière à être assez tolérant sur certains points, tel que notamment la lecture du fichier, qui est permise quel que soit le nombre de saut de ligne entre les niveaux, le nombre d'espaces, ou encore le codage des sauts de lignes utilisé (CR, LF, CRLF).

1.1 Description des structures de données utilisées

Les structures utilisées sont toutes regroupées dans le fichier « structs.h ». Les headers sont assez documentés pour connaître le rôle de toutes les variables des structures, nous ne nous y attarderons donc pas ici. Pour créer les structures du projet (et ce qui suit est aussi valable pour la création et la structuration des différentes fonctions du projet), j'ai procédé de la manière suivante : toujours partir du point de vue le plus global, pour ensuite approfondir et concrétiser les structures au fur et à mesure.

Il me fallait afficher une fenêtre qui contiendrait un Sokoban, j'ai donc créé la structure Sokoban qui contiendra toutes les données à afficher. Dans ce Sokoban, il me fallait afficher un niveau et des boutons, j'ai donc créé ces deux nouvelles structures. J'avais aussi besoin de connaître le mode de jeu et l'action faîte par l'utilisateur, ce qui se traduisit naturellement par encore deux nouvelles structures. On retiendra deux autres structures intéressantes : la map du niveau qui est un pointeur de pointeur sur une structure d'une case, qui sera en fait alloué dynamiquement comme un tableau à deux dimensions ; et une implémentation d'historique qui correspond à deux piles, qui seront utilisées simultanément avec les fonctions Undo et Redo. Concernant l'historique, mon raisonnement était le suivant : on pourrait sauvegarder à chaque action un instantané de la map, mais ce serait la façon la plus naïve de procéder, car cela impliquerait une surconsommation en mémoire (en théorie, mais pas sur si un si petit projet). Concrètement, à chaque action, trois cases au maximum peuvent changer : deux cases changent obligatoirement pour le déplacement du personnage et éventuellement une troisième, s'il y a déplacement d'une caisse. Pour chaque élément de l'historique, il suffit donc de stocker les différences qu'il y a à chaque action du joueur. Pour le déplacement du personnage, il suffit de stocker l'action qui a été réalisée par le joueur et de l'appliquer à l'envers. Pour le déplacement des caisses, deux pointeurs sur des cases sont présents dans la structure d'un élément : si aucune caisse n'a été

déplacée, les pointeurs sont à NULL, si une caisse a été déplacée, alors les pointeurs vont pointer vers les deux cases où il faut échanger la caisse.

Je vais aussi inclure les énumérations dans cette section. Ce ne sont pas des structures à proprement parler, mais elles permettent de créer certains types de variables qui prendront des valeurs définies par le développeur. J'aurais pu préférer utiliser des constantes à la place des énumérations, mais je n'avais, d'une part, nul besoin de connaître la valeur réelle de leurs états (que la valeur de l'action UNDO vale 3 ou 4, cela n'apporte rien à ma manière de procédé) et d'autre part, les énumérations permettent d'identifier directement un type de variable (par exemple, une action est une variable de type ACTION, plutôt qu'un int).

1.2 Description des algorithmes complexes

Dans cette section, nous aborderons trois points : l'algorithme de lecture dans un fichier, la manière de gérer l'historique et enfin l'algorithme de vérification de la fermeture de l'entrepôt.

Concernant la lecture dans le fichier, le but était d'avoir un stockage et un affichage adaptatif des niveaux, plus par défi et volonté d'apprendre que par réel besoin. Pour allouer seulement la mémoire nécessaire pour stocker un niveau, il fallait récupérer ses dimensions : la première étape de la lecture dans le fichier lit le niveau en entier et les calcule. On revient ensuite au début du niveau, grâce à une sauvegarde de la position du pointeur de lecture dans le fichier, faite avant le calcul des dimensions. On alloue la mémoire aux bonnes dimensions et on peut cette fois lire le niveau et le stocker. L'implémentation exacte est assez bien commentée dans le fichier source. On peut cependant noter que j'ai "découvert" que l'on pouvait directement, en accédant au pointeur « _IO_read_ptr » de notre descripteur de fichier, savoir ce qu'on allait lire avant même de le lire avec une fonction de « stdio.h ». J'ai utilisé cette technique pour la lecture, sans savoir si elle est recommandable. L'implémentation ne m'a pas particulièrement posé problème, elle m'a seulement pris un peu de temps car je ne programme pas des lectures de fichiers régulièrement.

Par ailleurs, je n'ai pas rencontré de difficulté à implémenter l'historique. J'ai d'abord écrit les primitives pour créer, ajouter, manipuler et supprimer une pile dans le module « historic ». Ensuite, j'ai créé les fonctions Undo et Redo qui manipulent un historique constitué de deux piles : à chaque action de l'utilisateur, une sauvegarde en est faite et empilée sur la pile Undo. La pile Redo est vide, jusqu'à ce que l'utilisateur fasse un Undo : on dépile la pile Undo, on traite l'élément pour le rétablir sur le niveau et on l'empile sur la pile Redo. Si on veut rétablir la dernière action, il suffit alors de dépiler le premier élément de Redo, de le traiter, puis de l'empiler sur la pile Undo. De cette manière, les deux piles nous servent à manipuler notre historique.

En revanche, l'algorithme permettant de vérifier que l'entrepôt est bien fermé m'a un peu posé problème. En premier lieu, j'ai dû choisir entre un algorithme itératif ou récursif : ce dernier m'a paru le plus adapté pour traiter des tests sur des cases dans un ordre non déterminé. Ensuite, il a fallu coder l'algorithme récursif, étape qui m'a demandé plusieurs heures de réflexions agrémentées de multiples tentatives non abouties. J'ai essayé une version naïve qui consistait à tester si les murs sont fermés. Je partais d'un endroit du mur, et testais s'il était fermé en faisant en sorte que la fonction se propage sur tous les murs adjacents de manière récursive. En théorie, c'était possible, mais compliqué à implémenter, car les murs peuvent faire des boucles, bifurquer (hors une bifurcation peut ne pas être fermée contrairement au reste du mur), ou encore y avoir plusieurs boucles de murs différents (par exemple, une boucle en haut à gauche et une autre boucle non reliée en bas à droite de la map). En résumé, il y avait trop de cas spécifiques qui compliquaient l'implémentation. J'ai donc réfléchi à un autre moyen de procédé : au lieu de tester les murs, il est possible de tester les cases qui ne sont pas des murs. Ainsi, l'algorithme se déroule comme suit : on part de la position du personnage et on se propage récursivement sur toutes les cases où le personnage peut aller. Si on arrive au bord de la map, c'est que les murs ne sont pas fermés et qu'on a systématiquement été bloqué par un mur. Cette solution combine plusieurs avantages : l'implémentation est beaucoup plus simple et l'on est sûr que le personnage est enfermé. En effet, avec l'algorithme précédent, les murs pouvaient être fermés sans pour autant que le personnage ne soit à l'intérieur. Pour ce qui est de l'implémentation exacte, toutes les étapes sont commentées dans les fichiers sources.

2 Regard final sur le code et le projet

2.1 Critique sur le code et améliorations possibles

Globalement, je suis assez satisfait de mon travail : je le trouve organisé, j'ai veillé à ce que toute erreur soit prise en charge, j'ai soigné au mieux le code et l'affichage et enfin le projet répond aux exigences du cahier des charges. Il reste néanmoins quelques points qui pourraient être améliorés. Premièrement, il m'est avis que le code du projet peut être optimisé et factorisé, car outre les headers relativement bien documentés, on compte environ 1700 lignes de codes dans les fichiers sources, ce qui me parait sensiblement élevé. Deuxièmement, les fonctions d'affichage des boutons et des informations sont à revoir, car la gestion des coordonnées et de la taille des éléments est un peu anarchique (certains nombres sont codés en dur, on a des constantes, define et variables éparpillées et pas forcément cohérentes). Concernant la fonction d'affichage des boutons, j'ai essayé de faire les choses correctement en utilisant des constantes et en ne codant aucun nombre en dur. Mais j'ai oublié de réfléchir à une manière de coder la fonction d'affichage des informations lorsque j'organisais l'affichage global du Sokoban. Somme toute, le développement a été mal organisé, ce qui se traduit par une fonction avec des ratios codés en dur pour organiser la taille d'affichage des informations. J'ai donc essayé de réaliser un affichage qui s'adapte aux dimensions de la fenêtre. Visuellement, je trouve ça plutôt réussi, sauf quand la fenêtre, et donc par conséquent les écritures, deviennent très petites, comme sur certains niveaux. Concernant cette partie du code, je pense qu'il faudrait trouver une autre technique pour gérer cet aspect du développement.

2.2 Questions suite au projet

En premier lieu, je me pose une question sur le Makefile, relative à une consigne que je n'ai pas appliquée : « une cible test qui doit effacer toutes les données des précédentes compilations, tout recompiler et exécuter le Sokoban ». Peut-être y a-t-il un intérêt que je ne distingue pas, mais selon moi, un des points forts de « make » (parmi tant d'autres) est son système de contrôle des dates des dépendances des cibles. Il permet, dans le cas où une compilation déjà présente est obsolète par rapport aux dépendances, de recompiler seulement les fichiers nécessaire automatiquement, sinon, il exécutera le Sokoban sans tout recompiler. Si on efface tout à chaque fois comme demandé dans la consigne, on perd la puissance de ce système. Par ailleurs, d'autres interrogations se sont présentées à moi tout au long du développement du projet. Je me suis demandé quelle était la bonne méthode pour afficher des éléments et des textes proportionnellement à la taille de la fenêtre ; si nous étions supposés libérer la mémoire de tout ce qui a été alloué avant un exit(EXIT_FAILURE) ; et s'il vallait mieux utiliser des tabulations ou des espaces dans notre code source.

3 Conclusion

Avec du recul, le projet m'a été bénéfique, non seulement par le code, mais aussi par bien d'autres aspects : j'ai appris à organiser un projet en sous-dossiers, écrire rigoureusement mes fichiers sources et headers, pratiquer de manière optimisée l'utilisation de GDB, utiliser Valgrind en ignorant les erreurs non causées par mon code, avoir un Makefile automatisé et complet, et j'ai découvert un bon nombre de fonctionnalités du formidable logiciel de contrôle de version Git. De surcroît, j'ai revu les bases du langage LaTeX, et découvert le Markdown utilisé pour les README sur GitHub (dans mon cas). Tous ces aspects, à côté du code, n'étaient pas forcément nécessaires pour ce projet-ci, mais cela me tenait à cœur et me servira sans aucun doute dans mes projets futurs durant les années a venir.