Rapport projet technologique
Groupe TM5Z

BANDET Alexis CAQUELIN Brieux
MANDALLENA Thibaut

11 Avril 2018

Table des matières

Rapport

1 Mise en place du projet

Contrairement à beaucoup de groupe, notre équipe ne s'est pas formée en octobre mais en janvier. Elle est le fruit de la fusion entre deux groupes. Le premier groupe, composé de BANDET Alexis et de CAQUELIN Brieux, a vu très tôt l'abandon de deux de ses membres. Le second groupe dont le seul rescapé est MANDALENNA Thibaut a subit un sort identique.

Dès lors, et avec l'autorisation des professeurs en présences, il a semblé judicieux de former un nouveau groupe et de laisser les absents de coté. Ainsi s'est déroulé la genèse du groupe TM5Z dont vous lisez le rapport de projet en ce moment même.

Il a donc été décidé de reprendre le travail du premier groupe, effectué de Septembre à Janvier, afin de mener le projet à son terme. Quelques ajustements dans l'organisation plus tard, le travail pouvait reprendre avec la nouvelle équipe.

Nous devions developer un clone du jeu undead de la collection de jeu de puzzle de Simon Tathams [1]. C'est un jeu de puzzle où le but est de placer des monstres, sur une grille, dans des cases ne contenant pas de miroir. On a trois type de monstres :

— Zombie qui sont toujours visibles
— Vampire qui ne sont visible qu'avant une reflexion dans un miroir
— Fantôme qui ne sont visible qu'après une reflexion dans un miroir

Chaque bord de grille contient un numéro, qui correspond au nombre de monstre que l'on doit voir en regardant par ce coté de la case.

Les miroirs peuvent être orientés dans deux sens différents, correspondant chacun a une diagonale.

1. https ://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/undead.html

2 Implémentation de la librairie "games"

2.1 Synthèse

Une seule librairie a été utilisée lors de ce projet. Cette librairie est créée à partir de deux fichier .c : *game* et *game_io.c*. Le fichier *game.c* gère le jeu en lui même. Tandis que le fichier *game_io.c* permet de charger et de sauvegarder des grilles de jeu. Toutes les instances du jeu, graphique, textuelle ou solveur utilisent cette bibliothèque.

2.2 Analyse

Dans un premier temps il a fallut définir la structure du jeu. Le choix s'est porté sur une structure contenant :

- Un tableau de tableau de tableau pouvant stocker les labels [2]. Il y aurait un tableau par direction [3].
- Un tableau contenant le contenu de toute les cases de la grille.
- Trois variables int qui stockent le nombre de monstre necessaire de chaque type (zombie, fantome et vampire) [4].

Ensuite, il a fallut implémenter les fonctions utiles et nécessaire au bon déroulement du jeu. Cela n'a pas été très difficile puisqu'il s'agissait surtout de coder, la reflexion ayant été faite en amont. La principale préoccupation était d'avoir un jeu fonctionnel au bout du compte. Il ne fallait donc rien oublier.

Le plus dur a été de penser la fonction permettant de parcourir la grille de jeu pour savoir combien de monstre était vu depuis un coté de la grille. C'était le point critique de l'implémentation de la librairie, après la definition de la structure. Il faut un algorithme rapide pour que le jeu ne gaspille pas de ressources. Dès lors il etait imperatif d'avoir une complexité linéaire par

2. Les labels sont les numéros sur les bords de la grille représetants le nombre de monstres devant être vu.

3. Une direction étant un coté de la grille. Dans le cas présent il y a quatre directions.

4. Le nombre de monstre étant souvent très faible on aurait pu prendre des variables de type short afin de gagner de la place en mémoire. Mais le peu de place gagner ne justifie pas l'utilisation de ce type de variables. De plus, cela n'aurait pas permit la fantasie de faire des grille très grandes.

rapport au nombre de case à parcourir. Ce fut la principale difficulté de l'implémentation de la librairie.

2.3 Bilan

La librairie *"game"* est developpée pour fournir des performances élevées du fait de son role central dans le déroulement du jeu. Les fonctions sont en temps constant, et toutes fonctionnelles. Il n'y a pas de fuite mémoire non plus, ce qui est particulièrement important quand un programme fait de nombreux appels de fonctions. La quantité de mémoire perdue pourrait alors devenir très importante au point de saturer la mémoire et de faire planter le systeme d'exploitation.

3 Extension

3.1 Synthèse

En cours de projet, il fallut faire face un "caprice de client". C'était une simulation de changement d'idée du gamedesigner alors que le projet était bien avancé. Il fallut donc s'adapter au caprice :

- rajouter un monstre SPIRIT qui n'a pas de réel impact sur le jeu en lui-même si ce n'est occuper une case. Puisqu'il n'est ni vu en direct ni vu avant ou après un mirroir
- rajouter des miroirs verticaux et horizontaux. Le but de cet ajout est de complexifier le jeu.
- la possibilité de changer la taille de la grille qui était alors fixée à quatre cases par quatre case. Les nouvelles dimensions pouvait être libre.

Le but de ces réclamations est d'evaluer la pérénité du code et d'avoir un regard critique sur l'utilité d'un code propre et bien pensé afin de s'adapter à ce genre de pratique.

3.2 Analyse

Le principal objectif, en plus de l'ajout de fonctionnalités, était de garder la compatibilité avec l'ancienne version du jeu.

Dans un premier temps, la structure du jeu est modifiée pour inclure une largeur et une longueur de grille (deux variables), ainsi qu'une nouvelle variable pour le nombre de monstre SPIRIT. Elle reste ainsi compatible avec l'ancienne version du jeu. Les fonctions de configurations de la grille de jeu ont été factorisé afin d'éviter la redondance de code. Il fallut juste s'assurer que lorsque l'on utilise l'ancienne version la grille était de largeur quatre par quatre et que le nombre de SPIRIT est initialisé à 0.

3.3 Bilan

Le code étant à la base propre et bien commenter, rajouter ces quelques lignes et fonctions n'a pas été difficile. Ce changement inclut donc des changement dans les fonctions test pour concorder avec notre optique de faire un développement dirigé par des tests (TDD)

3.4 Améliorations

Avec le recul il aurait sans doute été judicieux d'inclure une variable dans la structure pouvant stocker la version du jeu. En effet, dans notre code toutes les fonctions, à part celle de création de la structures, ont été modifié et non pas créées. Dès lors, la première version du jeu peut contenir des spirits en cas de bogue majeur (bien que la configuration fixe leur nombre à 0). En effet il est possible de placer un spirit et leur nombre est évalué par les fonctions. Un simple *short* avec le numéro de version aurait permit d'éviter cela en empechant le positionnement et l'évalutation de spirit et de miroirs horizontaux et verticaux.

La modification n'est cependant pas difficile au vu de la simplicité du code.

4 Le mode texte

4.1 Synthèse

Avec une librairie fonctionnelle il était possible de faire un jeu textuel, pouvant s'executer sur la sortie standard (en terminal). Les consignes de mise en forme étaient stricte, ce qui permit de ce concentrer uniquement sur le fond, à savoir uniquement respecter les consignes. L'utilisateur devait rentrer un coup sous la forme suivante : <**x**> <**y**> <**G**/**V**/**Z**>.

4.2 Analyse

Toutes les fonctions permettant le bon déroulement d'une partie étaient implémentées. Il manquait des fonctions pouvant afficher la grille et demander un coup à l'utilisateur.

La principale difficultée fut de demander à l'utilisateur de rentrer son coup. La mauvaise connaissance des librairies standards en C eu pour effet une perte de temps importante.

Autre problème, le fait de transcripter un Content [5] en charactère. Une fonction a donc spécialement été codé pour ce cela.

Enfin, il fallut definir une phase de jeu. C'est-à-dire mettre en place une boucle de jeu, qui va vérifier si le jeu est terminer, si le coup est possible etc... La première chose est de traiter la demande le coup du joueur Puis la traiter pour mettre à jour la grille. Ensuite il faut vérifier que le jeu est terminé, il y avait une fonction spécialement pour cela [6]. Si le jeu n'est pas terminé alors on recommence.

4.3 Bilan

Des fonctions d'affichage ont été implémentées, et le problème de l'intéraction avec l'utilisateur réglé. Encore une fois, les performances du programme était le centre de la reflexion. Il est imperatif que l'affichage du jeu se fasse rapidement, sans gaspiller de ressources, mémoire ou CPU. L'affichage se fait donc avec une complexité linéaire, dépendant encore une fois du nombre de case à afficher.

5. Un content est une case. Elle peut être vide (EMPTY), ou contenir miroir ou monstre.

6. fonction *isgame_ over()*

4.4 Améliorations

Les fonctions d'affichages ont été codé directement dans le fichier *undead_text.c*. Il aurait été plus judicieux de les mettres dans un fichier séparé par soucis de modularité et de lisibilité du code.

Si la grille est trop grande pour la sortie standard alors l'affichage n'est plus correct. Le problème est compliqué à résoudre, et le mode texte est principalement là pour tester la librairie. Il est donc connu mais n'est pas une priorité pour le moment.

Surtout, faire un fichier avec une boucle de déroulement d'une partie du jeu aurait été grandement utile pour la version graphique. Cela aurait éviter de refaire le même travail trois mois plus tard et aurait économisé du temps.

5 Solveur

5.1 Synthèse

Le solveur a été implémenté avec des structures pour décomposer le jeu de façon a pouvoir résoudre l'un après l'autre chaque labels en fonction des labels déjà resolu. Tout le corps de l'algorithme est dans la fonction main, et réagit differement suivant le premier parametre donnée à l'executable (FIND_ONE | NB_SOL | FIND_ALL). En respectant le cahier des charges, le résultat est enregistré dans un fichier au préfix spécifié par l'utilisateur.

5.2 Le corps main

La fonction main va dans un premier temps vérifier "rapidement" que le jeu a peut etre une solution. Pour ça il va d'abord vérifier que le jeu a autant de cases libres que de monstres à placer, après quoi il va tenter de trouver une solution pour chaque labels. Pour trouver une solution à un label ; 2 fonctions sont utilisé :

next_sol_for_label :
: Pour résoudre les labels sortants par un un autre label, dans ce cas les deux labels sont résolu ensemble, d'où le paramètre op_lab

next_sol_for_label_ret :
: Pour résoudre les labels aillant un mirroir vertical ou horisontal, dans ce cas le label fais un aller-retour

Ces deux fonctions sont similaires par leur procédé mais trop différentes pour etre compressible. Pour plus de détailles voir 5.3.
Si la vérification échoue les comportements diffèrent selon le choix du premier paramètre.

Après cette rapide vérification le tableau des structures de labels est réinitialisé et trié en ordre croissant avec un algorithme "bulle" en fonction de leur valeur, avec la fonction *labels_sort()*. Dès lors on rentre dans la boucle de résolution principale qui tourne sans condition de sortie, la méthode *break* est utilisée quand on a trouvé toutes les solutions.

Pour trouver la première solution l'algorithme va parcourir le tableau des labels en commencent à l'indice 0. On rentre alors dans une boucle qui va parcourir le tableau des labels de sorte à trouver la solution du jeu. Pour cela le label de l'indice est initialisé avec *refresh_frames()* (cf 5.4) pour obtenir la liste des structures des cases (*struct frame*). Après ça on cherche la solution suivante du label.

Dans le cas où on a trouvé la solution on incrémente simplement l'indice, en plaçant dans le jeu la solution du label, pour que le label suivant soit initialisé avec la solution trouvé. Dans le cas ou on a pas trouvé de solution suivante, on détruit les cases initialisées et on décrémente l'indice jusqu'a trouver un label avec des cases initialisé. Si on revient tout au début alors le jeu n'a plus de solution.

Cet algorithme prend aussi en compte la gestion des cases de jeu qui ne sont pas vue par les labels. Une autre boucle infinie permet donc trouver chaque solutions au cases indépendantes (initialisées par *get_useless_frames()*) et c'est dans cette boucle que le décompte des solutions ce fait pour NB_SOL, ainsi que l'enregistrement des solutions pour FIND_ALL.

5.3 Trouver les solutions d'un label

Les fonctions *next_sol_for_label()* et *next_sol_for_label_ret()* suivent le même schéma algorithmique.
On se sert de variables indiquant le nombre de fois qu'un monstre est vue suivant la caractéristique de la case. Par exemple, un fantome avant un mirroir pour le label principal vaudra 0 mais vaudra 1 pour le label opposé, car celui-ci voit le fantome après un mirroir. Pour la gestion du label oposé on initialise un dictionnaire enregistrant les indices correspondant des cases

de celui-ci avec les indice du label principal. Ensuite on ce place soit sur la dernière case du label si une solution a déjà était trouvé, soit sur la première.

Dans la boucle pour trouver une solution, on initialise les valeurs des monstres suivant les caractéristiques de la case pour le label ainsi que pour son oposé (si il y en a un), puis suivant le contenu de la case on essaie de placer un monstre. On incrémente si on a trouvé un monstre valable, on décrémente dans le cas contraire. Si on arrive au bout du label on a trouvé une solution, et si on arrive au début il n'y a plus de solution.

5.4 Réinitialiser les cases vides vue par un label

Pour réussir à trouver les solutions d'un jeu, il faut prendre en compte la solution trouvé du label précédent pour trouver celle du label actuel. Ainsi la fonction *get_label()* laisse le tableau des cases vide vue à NULL, c'est *refresh_frames()* qui s'occupe d'initialiser les cases vue par le label.

Dans une boucle qui tourne tant qu'on ce trouve à l'interieur du tableau de jeu, on va enregistrer chaques cases vides dans un tableau. De même on va initialiser le nombre de cases avant et après un mirroir, on initialise la valeur du label en la décrémentant quand un monstre est croisé, et on note chaque case vue en double par croisement de mirroirs (information enregistrée dans la structure de la case). La solution du label est pret à etre trouvée.

5.5 Amélioration ?

Une faiblesse de ce solveur est que l'on ne prend pas en compte le cas ou on doit trouver la solution d'un jeu avec certain monstre déjà placé, mais la correction est simple et par ailleur le solveur implémenté dans la version graphique prend justement en compte cela. De nombreuses améliorations dans la lisibilité du code pourrait etre aussi faites.

6 V2 en mode graphique

6.1 Rendu graphique du jeu

L'interface graphique a été implémenté en reprenant l'exemple donnée d'utilisation de la library SDL. Le fichier main.c a été simplement modifié pour afficher le logo de notre groupe quelques secondes avant de démarer le

jeu.
La structure graphique du jeu est composé des elements suivants :

- Le fond étoilé :
  La texture est copiée pour recouvrir toute la fenêtre en fonction de sa taille.

- Le logo en entête :
  il est placé simplement en haut à gauche, sa largeur est calculé en fonction de la largeur de la fenêtre.

- Les nombres de monstres disponibles :
  La texture est créée et copiée avant d'etre détruite dans la fonction de rendu principale en fonction du nombre de monstres disponibles. Le nombre de monstres est décrémenté suivant leur utilisation dans le jeu, et quand il est épuisé il s'affiche en rouge.

- Le tableau du jeu avec ses labels sur les 4 cotés :
  C'est la structure la plus complexe. Elle est composée des textures des cases suivant le contenu du jeu, quand la case est selectionnée on utilise les textures *_s*, quand elle est survolée on utilisent les textures *_under* et sinon on utilise les textures normales. Les mirroirs ne peuvent pas etre selectionnée ni survolée. Les textures des labels sont créée comme les nombres de monstres disponnibles dans le rendu du jeu (*game_rendre()*) ; elles s'affichent en cyan quand le label n'est pas complet, en vert quand il est bon, en rouge quand il n'est pas bon. Dans le cas ou l'indice est actif, la case selectionnée affichera le contenu de la solution sauvegardée dans **env->sol**.

- Le bouton restart :
  Il est placé en bas de la fenetre est centré. Il est affiché avec la texture *_under* si il est survolé.

- Le bouton indice et le nombre d'indices restant à coté :
  Il est placé en bas a droite, avec le nombre d'indices restant quelques pixels plus à gauche. Il est affiché avec une texture verte si il est activé et avec une texture rouge sinon. Quand il n'y a plus aucun indice plus rien n'est affiché.

Le jeu est chargé depuis un fichier txt si le nom est donnée en parametre, sinon un jeu prédéfinie (celui de game.h donnée en exemple) est initialisé.
La structure de l'environnement va inclure toutes les textures utilisées, le jeu, les coordonnées de la case selectionnée, les coordonnées de la case survolé par la souris, un booléen pour savoir si le bouton restart est survolé, un bouléen pour savoir si le mode indice est activé, le nombre d'indice disponible et la solution du jeu (cf 6.3). Le rendu est découpé en un rendu principal (*render()*), affichant le fond étoilé, l'entête, le bouton restart et le bouton d'indice. Quand le jeu est gagné on affiche le rendu *you_win()*, sinon le rendu *game_render()*.

6.2 Le control avec la fonction *process()*

La fonction de procédure permet de gérer les intéractions clavier et souris (pour un ordinateur) pour pouvoir intéragir avec le jeu. Cette fonction prend en paramètre une variable environement **e** pour gérer les différentes entrées de commandes.

Dans le cas d'une action souris, on va opérer différentes actions suivant l'emplacement du clique.

Si on a cliqué sur une case de jeu qui peut être modifié, on initialise les coordonnées selectx et selecty de l'environement. Si on a cliqué n'importe où ailleurs excepté sur le bouton indice on réinitialise ces valeurs à -1.

Si on a cliqué sur le bouton restart on restart le jeu avec *restart_game()*

Si on a cliqué sur la bouton d'indice on passe en mode indice (cf 6.3) et on décrémente le nombre d'indice. Dans le cas où on est déjà dans le mode indice, on le désactive. Par ailleurs cliquer n'importe où désactive le mode indice.

Dans le cas d'une action clavier, on va soit placer un monstre avec Z,V,G,S, et E pour enlever un monstre, soit ce déplacer avec les flèches directionnelles. Les flèches directionnelles permettent de ce déplacer d'une case modifiable à l'autre, si une case est déjà selectionnée. De même un monstre est placé uniquement si une case est selectionnée. Quand le mode indice est actif on

ne peux plus bouger la selection ou placer un monstre. La fonction prend aussi en compte l'emplacement de la souris, quand elle survole une case les coordonnées sont enregistrées dans **mousex** et **mousey** pour pouvoir afficher la case en *_over*, et de même pour le bouton restart quand il est survolé, la variable **restart_push** est actualisée.

6.3 Integration du solveur pour les indices

Pour pouvoir obtenir un indice, le solveur a été intégré au code. Quand le mode indice est activé, une solution est trouvé avec la fonction *find_solution()*. Cette fonction utilise la boucle de résolution dans la fonction main du solveur, en prenant en compte les monstres déjà placé pour apporter un indice correspondant au plateau de jeu actuel du joueur. Si aucune solution est trouvée un message est envoyé. Pour pouvoir utiliser un indice une case vide doit etre selectionnée, et dans ce cas l'indice est consomé, 7 indices sont disponibles au début d'un jeu. Le bouton restart réinitialise le compteur. Quand le mode indice est quitté la solution est détruite.

Le code ayant était simplement copié collé pour les fonctions et les structures permétant la résolution, on pourrait envisager d'inclure dans la library les fonctions de résolution pour éviter d'avoir des doublons dans chaques fichiers et ainsi rendre le code bien plus propre.

7 Test & debug

7.1 Synthèse

Un travail de test plus approffondi de test et de debuggage a été effectué en fin de projet. Il avait pour but de corriger des erreurs ou des abérations non detectées en amont.

Plusieurs outils de test de couverture, de profilage et d'analyse statique ont été mis à contribution.

7.2 Test de couverture

Un test de couverture est effectué à l'aide de l'outils **gcov**. Pour cela il a fallut modifier le fichier *CMakeList.txt* afin de compiler le programme avec les paramètres necessaires.

Le principe du test de couverture est de vérifier que nos tests executent bien toutes les lignes du programmes. Ça ne prouve pas que le programme est bien testé, mais permet de voir ce qui n'est *a priori* pas testé.

Ensuite, les commandes console **"make test"** et **"make ExperimentalCoverage"** permettent de vérifier si des lignes de codes n'ont pas été exécuter lors des tests. Cela a permi de constater par exemple que la fonction *copy_ game()* n'est pas testé lorsque son parametre est *NULL*. Ainsi ces lacunes furent corrigées.

7.3 Analyse statique de code

Un test d'analyse statique du code a été effectué à l'aide de **cppcheck**.

Comme son nom n'indique, l'analyse statique de code analyse le code sans l'executer. C'est une analyse des erreurs classiques.

Les commandes
cppcheck –enable=all –xml-version=2 -I ../include *.c 2> report.xml
et
cppcheck-htmlreport –source-dir=. –report-dir=report –file=report.xml
permettent d'obtenir une analyse statique au format html. Il suffit alors de lire l'analyse sur un navigateur.

Des erreurs, comme des variables non utilisée, des erreurs de code comme une attribution de valeur à la place d'une comparaison de deux variables [7], ont été corrigés.

7.4 Optimisation et profilage

L'optimisation du code a été testé l'outils **callgrind** du programme valgrind.

Ce dernier permet d'optenir un rapport complet de l'execution d'un programme. Notamment, il peut detecter les fonctions demandant le plus de ressources, ou les plus fréquemment utilisées.

Aucune anomalie ne semble s'être glissé dans le code. Et aucune fonction ne semble prendre plus de ressources que les autres.

7. Il y avait un "==" à la place d'un "=". Pourtant aucun warning lors de la compilation.

7.5 Valgrind

Enfin, le programme **valgrind** permit tout au long du projet de detecter les fuites de mémoire. Cela assure que le programme désalloue et alloue le même nombres de place en mémoire entre le début et son exécution et la fin de son exécution.

Comme précisé plus haut dans le rapport, cela pour devenir critique lorsque que programme alloue et désalloue souvent de la mémoire.

7.6 git

L'outils **git** n'a pas vraiment été utilisé comme un gestionnaire de version, mais plus comme un outils de partage. Il n'a jamais servit a revenir en arrière dans les commits. Il permettait à n'importe qui dans l'équipe de pouvoir télécharger les derniers fichiers. Eventuellement de voir les modifications, même si dans la pratique cela n'a pas été utilisé.

C'est un outils très puissant pour les travaux de groupes mais il n'a pas été exploité à sa juste valeur.