

BringBackMoney

-->ENTREE POUR JOUER<--

(c) COMMANDE | (a) ABOUT | (q) EXIT_

BringBackMoney (jeu ASCII)

RAPPORT DE PROJET EN PROGRAMMATION STRUCTURÉE

Simon Marollerau | SE2A3 | 01/02/2022



Sommaire :

I - Introduction et présentation du jeu.....	2
II - Initialisation du monde et affichage graphique.....	3
III - Mouvement et interaction.....	5
IV - Le menu et la sauvegarde	7
V - Difficultés rencontrées	9
VI - Optimisation futur	11
VII - Conclusion	11

I - Introduction et présentation du jeu

Dans le cadre du module programmation structurée. Nous avons dû créer un jeu en langage C à partir d'un cahier des charges fourni. J'ai ainsi codé le jeu sous Visual Studio Code avec le compilateur MinGW sur Windows. J'ai associé mon projet avec un dépôt distant sur github où vous pouvez retrouver toutes les versions du projet.

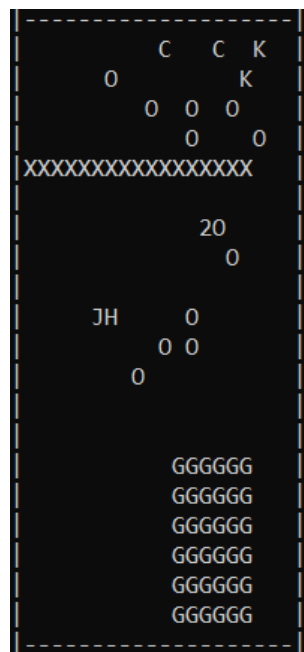
Bring Back Money est un jeu où vous interprétez le joueur correspondant au caractère 'J' dans une carte en 2 dimensions. Le but est de ramasser le plus de pièces tant que c'est possible, car des pièges ou encore des monstres sont présents dans vos alentours. Cachez-vous dans les buissons et certains monstres ne vous poursuivront plus. N'oubliez pas aussi de passer par votre cabane afin d'y déposer un maximum de pièces. Si vous finissez par perdre tous vos points de vie, le jeu s'arrête, vous demande votre nom d'aventurier et sauvegarde votre score.

Le but de ce projet est de mettre en application les différents concepts que nous avons appris à maîtriser durant ce module.

II - Initialisation du monde et affichage graphique

Le monde est stocké dans un tableau à deux dimensions déclarées avec les valeurs max (20 par 20). Ensuite le monde est rempli petit à petit de tous les éléments un par un, allant des obstacles jusqu'au placement de joueur en passant par le remplissage des buissons, des pièces et même les monstres. Tous les éléments du jeu sont générés pseudo-aléatoirement grâce à la fonction *alea(min,max)*, qui va générer un nombre aléatoire en fonction d'une graine qui sera différente (le temps). Ces fonctions aléatoires vont ainsi aider à générer la position d'un élément mais aussi le nombre d'éléments qu'il faut faire apparaître sur la carte. Chaque élément apparaît normalement dans un espace vide sur la carte pour ne pas interférer avec les autres éléments. Tout ce processus est ainsi regroupé dans le header file nommé *init_carte.h*. Les fonctions procèdent tous à peu près de la même manière :

- On random un certain nombre d'éléments.
- On random les positions y et x tant que celles ci ne correspondent pas à un éléments vide de la carte ou on vérifie qu'il n'y à pas déjà un certain éléments sur cette case.
- On les positionne aléatoirement dans la carte à l'aide de deux boucles qui vont parcourir tout le tableau en X et en X.



Carte 2D généré pseudo aléatoirement

Vu que le joueur et les monstres ont plusieurs variables telles que la vie du joueur, son nombre de pièces, on peut les considérer comme des "objet", ces éléments particulier

nous les avons définis à l'aide de structure qui regroupe un certain nombre de variables. Même si le cahier des charges proposait de les gérer avec des tableaux j'ai décidé de le faire avec des structures après quelque recherche et afin de simplifier la compréhension de mon code. Nous avons donc notre structure pour notre Joueur, néanmoins pour les monstres c'est différent nous en avons plusieurs à gérer, pour pallier ce problème nous avons donc fait un tableau de structure regroupant tous les monstres, il suffira juste de boucler leurs interactions qu'on verra plus tard afin de le faire pour tous les monstres.

```
struct Squelette_Monster {  
    int pos_x;  
    int pos_y;  
    int type;  
    int NbMonstre;  
    char on_object;  
};
```

```
struct Squelette_Player {  
    int life;  
    int nb_key;  
    int pos_x;  
    int pos_y;  
    int poscab_x;  
    int poscab_y;  
    int coins;  
    int cabane_coins;  
    int J_buissons;  
    int J_cabane;  
};
```

Structure représentant les variables composant un Joueur et un Monstre

Les fonctions d'affichage permettent elles de récupérer les informations de la carte et de les afficher sur le terminal. On a donc 3 affichages : l'affichage de la carte, l'affichage de l'interface cabane quand on rentre dans la cabane, et l'affichage des informations du joueur en jeu qui sont juste en dessous le joueur. L'affichage de la carte et celle de l'interface joueur se refresh à chaque tour car ceci équivaut à une évolution dans la carte comme un déplacement du joueur. La petite subtilité de l'affichage de la carte provient des éléments pièges noté comme un 'P' dans notre carte. Vu que ce sont des pièges dans notre tableau à deux dimensions ils sont bien notés 'P' mais dans l'affichage nous remplaçons le 'P' par un espace vide (' ') pour que le joueur ne voit pas les pièges !

Les autres affichages, plus secondaires, interviennent à des points précis comme celui de la cabane, lorsqu'on rentre dans cette cabane nous pouvons ainsi poser un certain nombre de pièces souhaité par l'utilisateur. L'autre c'est les affichages en fonction de l'interaction qu'on a avec certains éléments. Par exemple, si le joueur se prend un piège, un message sera affiché avec écrit, "Oh non tu as perdu une vie."

```
Bienvenue dans votre interface cabane !  
Vous avez actuellement 7 pieces sur vous  
Vous avez actuellement 0 pieces dans votre cabane.  
  
Appuyez sur 'e' pour quitter la cabane.  
Appuyez sur 'p' pour ajouter des pieces.  
  
Combien de pieces voulez vous déposer ? :
```

Interface cabane

Pour finir, l’affichage global du jeu n’est pas sur le terminal de base mais sur un terminal utilisé par la librairie Ncurses (curses.h) qui permet d’avoir une meilleur fluidité de la fenêtre, un contrôle sur chaque coordonnées de la fenêtre et permet aussi le mouvement dynamique (sans retour chariot) du joueur.

III - Mouvement et interaction

Le jeu se déroule en tour par tour c’est-à-dire que à chaque fois que le joueur fait un mouvement ou interagit il se passe un tour et l’environnement bouge. En algorithmique ça donne cela :

Début :

On initialise la carte

Tant que le joueur à de la vie :

On clear le terminal

On affiche la carte

On fait bouger le joueur

On fait bouger le/les monstre

Fin Tant que

Affichage Game Over

Fin

Cette boucle est le cœur du jeu, elle permet de se répéter tant que notre joueur possède de la vie. En réalité le “On fait bouger le joueur” est divisé en 2 parties. Dans un premier temps on récupère le caractère entré par l'utilisateur et on calcule la futur position du joueur en fonction de si il va en haut, en bas, à gauche ou encore à droite. Puis on observe sur quoi va interagir le joueur sur sa prochaine position, si c’est un espace alors il

se déplace juste, si c'est un obstacle alors il ne bouge pas, si c'est une pièce alors le joueur récupère une pièce et se déplace, si c'est un monstre alors le joueur perd une vie etc...

L'interaction entre le monstre et le joueur comporte deux fonctions (*interaction_monstre_joueur()* et *interaction_joueur_monstre()*) car nous avons deux cas : le cas où le joueur va sur la case du monstre et le cas où le monstre va sur la case du joueur. Pour le premier cas (*interaction_joueur_monstre()*), il faut juste vérifier en plus à quelle monstre on a affaire afin de savoir lequel modifier. Pour cela on boucle notre tableau comprenant tous les monstres et on cherche lequel correspond aux coordonnées actuelles du Joueur et lorsqu'on trouve le bon on le rattache à *interaction_monstre_joueur()* avec la bonne indice

```
void interaction_monstre_joueur(char carte[SIZE_Y][SIZE_X], s_player *Joueur, s_monster TableMonstre[MAX_MONSTER], int i) {
    //Interaction entre monstre et Joueur
    RANDOMIZER_SEED;
    int nb_random_x = alea(2,18);
    int nb_random_y = alea(2,18);
    do {
        TableMonstre[i].pos_x = nb_random_x;
        TableMonstre[i].pos_y = nb_random_y;
    } while(carte[nb_random_y][nb_random_x] != ' ');
    carte[nb_random_y][nb_random_x] = TableMonstre[i].type + '0';
    if((TableMonstre[i].type == 5 || TableMonstre[i].type == 6 || TableMonstre[i].type == 7 || TableMonstre[i].type == 8) && Joueur->coins>=1)
        Joueur->coins -= 1;
    else {
        Joueur->life -=1;
    }
}

void interaction_joueur_monstre(char carte[SIZE_Y][SIZE_X], s_player *Joueur, s_monster TableMonstre[MAX_MONSTER]) {
    for (int i = 0; i < TableMonstre[0].NbMonstre; i++) {
        if(Joueur->pos_y==TableMonstre[i].pos_y && Joueur->pos_x==TableMonstre[i].pos_x) {
            interaction_monstre_joueur(carte,Joueur,TableMonstre,i);
        }
    }
}
```

Fonctions d'interaction entre le monstre et le joueur et vice versa

Après avoir fait bouger le joueur "On fait bouger le/les monstres ", Les fonctions de mouvements sont ressemblantes à celles du joueur, Néanmoins les monstres ne bougent pas par rapport à des entrées définies telles que les touches du clavier, mais à grâce encore à la fonction *alea(min,max)* qui va pouvoir générer des déplacements aléatoires pour les monstres en suivant bien évidemment des règles préétablies dans le code. Ces mouvements sont différents en fonction du type de monstre. Il y en a 8, les monstres de type 1 se déplacent de 1 case en 1 case aléatoirement et le monstre de type 2 se déplace dans un rayon de 3 cases maximum, ce sont des monstres "débiles". Les monstres de types 3 et 4 suivent le joueur avec une fonction qui calcule le chemin le plus court entre lui et le joueur, ce sont des monstres "intelligents". Les monstres 5, 6, 7 et 8 sont les mêmes que ceux d'avant mais au lieu de voler de la vie ils volent de l'argent au Joueur. C'est aussi là qu'intervient le deuxième

cas parler dans le à chaque mouvement du monstre et ça pour tous les monstres on check si on passe sur le joueur alors on rentre dans cette *interaction monstre joueur()* directement et exécute les choses à faire, c'est-à-dire qu'il enlève une vie ou une pièces au joueur en fonction de son type puis il se téléporte aléatoirement sur la carte pour ne pas rester à côté du joueur.

IV - Le menu et la sauvegarde

Après avoir programmé la partie jeu, j'ai pu organiser ce qu'il y a autour c'est-à-dire un menu, puis une sauvegarde dans un fichier .save dans le but de pouvoir comparer le score de chaque partie.

Lorsqu'on lance le jeu on accède tout de suite au menu du jeu, qui comporte 4 options, Soit on quitte le jeu, soit on joue qui sont les principales. Puis 2 autres options afin d'avoir une description du jeu et de ces éléments et l'autre pour voir les commandes du jeu. Ces 2 options ont une option unique de retour afin d'accéder au menu principal. Dès que le joueur n'a plus de vie alors la partie s'arrête et un écran de fin avec un texte vous indique que c'est fini et vous demandera d'appuyer sur entrée pour revenir au menu. Ainsi le jeu est bouclé et l'unique méthode pour sortir est d'appuyer sur "ECHAP" ou de "ALT+F4" (comme un rajeu).

```
Commande de mouvement :  
- Z : HAUT  
- Q : GAUCHE  
- S : BAS  
- D : DROITE  
  
Commande d'interaction':  
- I : Interagir avec certains objets.  
- P : Insérer des pièces dans la cabane.
```

Pages des "Commandes"

```
Bring Back Money !  
Recuperez un maximum de pieces avant de mourir  
interagissez avec un monde rempli de caracteres !  
Description :  
- 'J' : Le joueur qu il faut deplacer sur la carte.  
- 'O' : Les pieces a collecter (collecte automatique).  
- 'C' : Des coffres qui peuvent soit contenir soit des pieces, soit des pieges.  
- 'K' : Des cles pour ouvrir les coffres.  
- 'G' : De l herbe qui permet de se cacher d un monstre intelligent.  
- 'X' : Un obstacle qui empêche la bonne circulation du joueur.  
- 'H' : Le spot de depart du joueur. Le joueur peut stocker des pieces dans la  
cabane sans que les monstres puissent lui voler.
```

Pages du "À propos"

Si on reprend l'algorithme précédent il faut alors rajouter :

Début :

Affichage Menu

Si on veut jouer :

On initialise la carte	}	Algorithme précédent
Tant que le joueur à de la vie :		
On clear le terminal		
On affiche la carte		
On fait bouger le joueur		
On fait bouger le/les monstre		
Fin Tant que		
Affichage Game Over		
Sauvegarde les résultats du joueur		
Retour Menu		

Si on veut voir les commandes :

Affichage des commande

Retour Menu

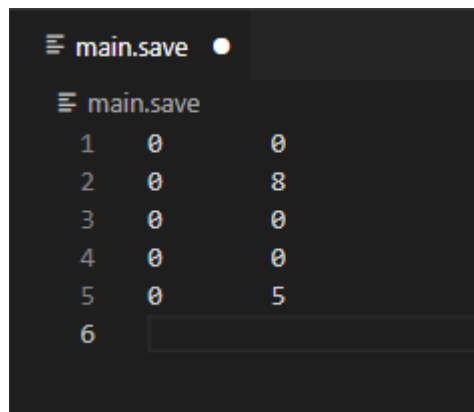
Si on veut voir le a propos :

Affichage du à propos

Retour Menu

Fin

On remarque une chose en plus en gras : “Sauvegarde les résultats du joueur”. En effet la fonction de sauvegarde intervient juste après le Game Over et permet d’enregistrer le score du joueur c’est-à-dire le nom de pièces qu’il a dans sa cabane. La librairie *stdio.h* est implantée de base au langage C, ayant des fonctions de gestion (*fopen()*, *fclose()* ...), j’ai pu les utiliser afin d’ouvrir le fichier *.save* écrire les informations et fermer le fichier.



```
main.save
main.save
1  0  0
2  0  8
3  0  0
4  0  0
5  0  5
6  
```

Contenu du fichier *main.save*

On observe ici qu'il y a eu 5 parties jouées et que sur la deuxième le joueur a réussi à récupérer 8 pièces et 5 pièces pour la dernière.

V - Difficultés rencontrées

Tout au long de ce projet j'ai dû faire face à de nombreux problèmes car avant de se pencher sur le code et les méthodes de programmation il fallait un environnement convenable pour débiter ce projet. Étant habitué à de la programmation web avec Visual Studio Code j'étais parti sur cette IDE. Mais trouver un compilateur windows pour du C c'était plus compliqué après quelques recherches et tutos j'avais réussi à enfin avoir un environnement convenable pour programmer.

Le second problème rencontré c'est la librairie Ncurses qui est inclus directement sur linux, il était simple de pouvoir l'utiliser. Néanmoins pour programmer sur Windows c'était autre chose. J'ai dû trouver un moyen d'installer une librairie portable de Ncurses, PDCurses via le programme d'installation de librairie du compilateur (*mingw-get.exe*) qui est caché dans ces fichiers.

```
* 038ff54 [fix] remplissage_buissons
* 3ac2bc1 [update] update exe
* ff931f4 [new] Ajout de nouvelle roadmap
* c97838a [update] update exe
* 8a4b78b [update] check_mark sur le README
* a12f22e Merge branch "Curses_integration" into develop

* 0b55649 (origin/Curses_integration) [update] RoadMap README.md
* ec75f6c [new] changement du nom du dossier
* 1b88983 [update] last version of exe
* 10b0cbc [fix] Refonte du switch case des déplacements + interaction_environnement()
* 5b89230 [new] remplissage_buissons()
* ec6013a [new] Ajout pour savoir si le joueur est dans les buissons ou non
* d0d5deb [fix] .exe
* d182c41 [update] RoadMap README
* 5dd5b1c [new] test check_interaction -> projet.c
* 3744567 [syntax] README
* ac32681 [syntax] README
* 1b0e90e [syntax] README.md
* b58c30b [syntax] README
* d00622d [new] intégration du curses + Mouvement dynamique
* a0f5b16 [last fix] README
* eacc25a [fix] syntax README
* d181695 [new] syntax README
* 7572467 [fix] syntaxe de README
* 9a700d3 [new] nouvelle roadmap
* 79fc7ec [NEW] test getkey()
* dbed1c8 [new] Begin Curses Integration test
* c058128 [fix] delete curses.h from develop
* a153111 [fix] Delete test curses from develop
* 3be2d87 [fix] syntaxe

* 02b5353 [fix] ubuntu
* a196f6d [fix] syntaxe
* e02985e [new] init_carte.h pour simplifier l'init de la carte
```

Création de la branche "Curses_integration" et fusion avec "develop"

D'autres difficultés ont été rencontrées mais plus sur l'aspect code que environnement. Un peu relié au problème précédent c'est bien évidemment l'apprentissage de la librairie ncurses qui a été problématique vu le peu de documentation officiel avec des exemple présent sur le net. Mais aussi sur la partie programmation du Joueur, comme la gestion des pointeurs avec une structure que je ne connaissais pas et qui m'ont posé du fil à retordre.

Pour finir, le plus gros problème a été lors du développement des déplacements de mes monstres. Après avoir commencé à intégrer interactions, mes monstres ont commencé à faire des déplacements qui n'étaient plus vraiment aléatoires c'est-à-dire qu'il foncé en diagonal puis il foncé tout à gauche puis il bougé plus et après ça partait vers le haut etc... J'ai retourné le problème dans tous les sens par rapport à la seed du random qui pouvait être mal déclarer, les déplacement des monstres mal codé mais finalement non, j'ai du alors revenir sur une ancienne version de mon programme créer un branche pour redévelopper en parallèle le déplacements et les interactions pour finalement refusionner avec ma branche "develop". Malheureusement la fusion de branche à buggé et j'ai dû recommencer tout le cheminement une deuxième fois, ce qui finalement à fonctionner !

```
* 7b09ea7 [.exe] update
* 1348f84 [fix] ajout de &Joueur dsans Type_Monstre()
* 6f8dda0 [new] dvp interaction_mstre_joueur
* 6e4bdce (origin/develop_fix_monster) [fix] interaction_joueur_monstre
* d30fb4e [fix] monster
* e084ec3 [.exe] update
* 3ce7d68 [fix] Optimisation Déplacement joueur et monstre + create interaction monstre_joueur
* 156d280 [fix] reaffichage buissons et piege
* 5e9bc9f [new] Meilleur affichage pour coord monstre
```

Log des commits de git pour le problème des monstres

VI - Optimisation futur

Ce projet nécessite encore quelques évolutions afin d'avoir un jeu encore plus optimisé et plus évolué. La proposition d'inclure un brouillard, ou encore une map ouverte qui évolue en fonction de où va le joueur est très intéressante je trouve. Même si la sauvegarde est intégrée au jeu, je n'ai pas encore codé les moyens pour l'interfacer au jeu comme un leaderboard afin de pouvoir comparer le score de chaque partie. J'avais commencé aussi à intégrer les variable pour le settings des paramètres simple avec une structure qui récupère les données et les envoie du plusieurs fonction, malheureusement par manque de temps elle n'est pas fonctionnel mais les fonctions se retrouve dans le code.

Nous aurions pu aussi créer un système de difficulté avec 3 choix facile, moyen, difficile avec derrière les proportions des monstres des pièges qui changerait. Bien évidemment beaucoup de gens le diront mais le C n'est pas le meilleur langage pour faire des jeux nous pourrions aussi envisager plus tard de migrer sur des langages plus orienté objet comme le C++ par exemple. Plus nous ferons évoluer le jeu, plus on sera limité à ce que propose le C. Même si une migration vers une meilleure interface graphique tel que SDL ou encore OpenGL est envisageable. Mais je préfère lui garder une identité old school avec cet affichage terminal.

VII - Conclusion

Finalement la version 1.0 du jeu fonctionne correctement même si encore quelques bugs peuvent survenir. Ce projet représente bien pour moi la finalité de ce module, il est le reflet de ce que j'ai pu apprendre au cours des 4 derniers mois. Ayant déjà codé en C auparavant ce module m'a permis d'acquérir une plus solide expérience sur ce langage et ce que nous pouvons faire avec. Néanmoins c'est le premier projet en C aussi conséquent que j'ai réalisé, et j'ai pu découvrir de nombreuses choses qui entoure un développeur tel que la gestion de version avec git, gérer un dépôt distant avec github, réfléchir sur l'optimisation et la complexité des fonctions. Peut-être une version 2.0 verra le jour