Rapport du Projet Mapio

Sommaire :

Introduction :

Notre groupe de projet est composé de LEONI Laëtitia, AHMAD BOISSETRI Binzagr et GHIBERT Lucas. Nous avons travaillé sur une version du jeu Mapio fournie par l'Université. Notre tâche était divisée en deux grandes parties.
Dans la première, nous avons dû permettre la sauvegarde et le chargement des cartes dans le jeu. Nous avons aussi fourni un utilitaire avec son lot de fonctionnalités. Parmi elles, l'accès aux informations de la carte sauvegardée, la modification de la taille de celle-ci ou encore la modification des types d'objet se trouvant à l'intérieur.
Dans la seconde partie, nous avons géré des temporisateurs. Toutes actions du jeu nécessitant un "timer" pour s'effectuer (clignotement de Mapio, explosion de bombes, etc ...) ont été résolues dans cette partie.

I. Première partie

1. Résultats :

Toutes les fonctionnalités demandées dans la première partie du projet ont été implémentées. Aucun bug n'a été trouvé au cours de leur utilisation.

Les fonctions et commandes qui s'effectuent correctement sont :

- map_save (mapio.c)
- map_load (mapio.c)
- getwidth (maputil.c)
- getheight (maputil.c)
- getobjects (maputil.c)
- getinfo (maputil.c)
- setwidth (maputil.c)
- setheight (maputil.c)
- setobjects (maputil.c)
- pruneobjects (maputil.c)

2. Choix et protocoles :

- map_save (mapio.c)

Lorsque l'on sauvegarde, on ne conserve pas les cases vides (affectées MAP_OBJECT_NONE). Cela rend la tâche plus rapide et donne un fichier .map moins lourd. Ce choix se répercute dans la fonction map_load qui a donc moins de données à lire et peut s'effectuer en un temps moindre aussi.
On sauvegarde en premier la largeur, la hauteur et le nombre d'objets différents disponibles dans l'éditeur de niveau puis, dans l'ordre, l'abscisse, l'ordonnée et le type (sous forme d'entier) de chaque objet.
Nous avons aussi choisi d'ajouter un entier -1 à la fin de tous les objets enregistrés dans le fichier de sauvegarde. Ce "marqueur" est utilisé lors de la lecture et du chargement de la carte pour différencier toutes les instances des objets de leur "définition" située à la fin du fichier .map.
Cette "définition" d'un objet correspond ainsi au chemin relatif de son fichier image, ainsi qu'à ces caractéristiques propres, soit une série d'entiers booléens. Nous avons rajouté cependant un entier indiquant la longueur de la chaîne de caractères du chemin relatif avant celui-ci afin de faciliter sa lecture lors du chargement ultérieur de la carte.

- map_load (mapio.c)

Lors du chargement, on lit les trois premiers entiers du fichier pour nous indiquer la taille de la carte à allouer et le nombre d'objets qui seront disponibles sur celle-ci.
Vient ensuite une boucle qui s'arrête à la lecture de notre marqueur -1. Elle lit puis place sur la carte (encore à l'état de simple matrice d'entiers) toutes les instances d'objets sauvegardés. En reprenant la fonction map_new, nous avons utilisé map_object_begin() pour démarrer l'implémentation des "définitions" des objets et ainsi les associer à l'entier qui indiquait leur type dans la matrice de la carte jusqu'alors.
Pour le chemin relatif des fichiers images des objets, nous avons alloué une chaîne de caractères de la taille indiquée par l'entier enregistré avant celle-ci. Nous avons aussi ajouté une case à ce tableau pour y insérer un caractère '\0' indiquant à la machine la fin de la chaîne de caractères.

- getwidth / getheight / getobjects / getinfo (maputil.c)

Ces quatres fonctionnalités sont regroupées en une seule fonction qui affiche les informations demandées par l'utilisateur par rapport à l'argument passé dans la console de commandes.
Une simple série de conditions et de déplacement du curseur dans le fichier nous a permis de répartir de façon optimale quelles informations sont données et quand.

POUR LA SUITE :
Pour toutes les fonctions impliquant la modification du fichier .map, nous avons décidé d'en créer un temporaire, d'y écrire l'ensemble de l'ancien fichier mais avec toutes les modifications nécessaires apportées et enfin de remplacer cet ancien fichier par le nouveau.
Ainsi lorsqu'une erreur se produit, ou qu'aucune modification n'a finalement besoin d'être apportée, nous supprimons le fichier temporaire, affichons un message d'explication et arrêtons le programme. Notre fichier .map original est donc ainsi préservé en toute circonstances.

REMARQUE :
Si l'utilisateur possède sur sa machine un fichier au nom correspondant exactement à celui dont on se sert pour effectuer les changements, les résultats sont inconnus. Il sera aussi effacé une fois le programme terminé.
Nous avons estimé ce risque minime car nous créons un fichier temporaire avec l'extension .map, particulière aux fichiers contenant la carte de notre jeu et que l'on a donc peu de chances de retrouver chez l'utilisateur un fichier similaire.

- setwidth / setheight (maputil.c)

Lors d'un agrandissement de la carte, que ce soit en largeur ou en hauteur, nous avons pris le parti de rajouter autant de mur et de sol que nécessaire pour préserver le cadre original du jeu. Nous décalons aussi en fonction le mur de droite qui constitue le cadre du jeu.
De même pour un rétrécissement, cette fois uniquement en largeur, le mur de droite vient épouser la nouvelle largeur maximale entrée par l'utilisateur. Le reste des objets se retrouvant en dehors des nouvelles dimensions de la carte est cependant "détruit" car non enregistré lors de la sauvegarde.

- setobjects (maputil.c)

Cette fonction réécrit entièrement les "définitions" des objets disponibles dans le jeu. Pour cela, une seule chaîne de caractère est utilisée pour recevoir les arguments de la commande. Elle est comparée à des chaînes de caractères fixes (ex : "collectible", "not-collectible") et c'est suite à cette comparaison qu'un entier booléen reçoit l'information concernant l'objet traité.
Toutes les informations ainsi recueillies sont ensuite écrites dans le nouveau fichier avec les nouveaux chemins relatifs des fichiers images et leur nouvelle longueur. Une fois ceci fait, on change le nombre d'objets disponibles dans cette nouvelle version du jeu si besoin est.

ATTENTION :
Avec l'implémentation que nous avons choisie, le changement dans l'ordre des définitions des objets n'est pas répercuté sur le reste du fichier. Par exemple, si l'on inverse les "définitions" des objets flower.png et coin.png, toutes les fleurs du jeu deviendront des pièces et inversement.
Si on supprime une "définition" d'objet, la "définition" qui prendra sa place dans l'ordre d'entrée des arguments verra ses instances remplacer celles de l'ancienne en jeu.
Enfin, si on supprime une "définition" d'objet qui a une instance dans la carte sauvegardée, cette instance sera remplacée par une instance de la dernière "définition" rentrée lors de l'écriture de la commande.

- pruneobjects (maputil.c)

pruneobjects retire du jeu toutes les "définitions" d'objet inutilisées.
On commence par une recherche des objets présents sur la carte du jeu. Pour cela, nous avons initialisé un tableau d'entiers à 0. Lorsque l'on rencontre un objet utilisé, sa case passe à 1. A la fin du parcours de la carte, si une case du tableau est resté à 0, il faut supprimer au moins une "définition" d'objet. On compte ensuite combien de type d'objets il restera une fois les objets inutilisés supprimés.
On parcourt la liste de "définitions" en fin de fichier. Si une "définition" correspond à un objet inutilisé, on ne la réécrit pas dans notre nouveau fichier.
Enfin, on décrémente l'entier correspondant au type des objets dont les "définitions" se situaient après celle(s) détruite(s) dans l'ancienne liste. On met aussi à jour le nombre d'objets différents disponibles.

II. Deuxième partie :

- struct echeance

Dans l'idée, la structure a été créée afin d'effectuer le traitement de plusieurs signaux. On y retrouve notamment le temps où le signal a été lancé (timer), le paramètre (param) qui doit être utilisé lors d'un sdl_push_event et un booléen (arme) qui indique si l'événement a été lancé.
On lancera ensuite un tableau d'échéanciers afin de tenter de gérer plusieurs SIGALRM.

- hand

C'est ici que l'on veut traiter tous les SIGARLM envoyés.
On récupère le temps actuel et celui du premier signal envoyé (qui est donc à l'indice 0 du tableau d'échéanciers). On initialise une variable booléenne (all_event_finished) à false et on met en place un while(all_event_finished == false).
Par la suite, on veut récupérer à chaque fois le premier événement (en prenant l'indice) qui doit arriver et n'a pas encore expiré, lancer sdl_push_event de celui-ci et indiquer que cet événement a expiré. Enfin, on vérifie si tous les événements ont expiré: dans ce cas, on sort de la boucle.

- demon

Ici il s'agit de la tâche qu'effectue le thread demon. On crée une sigaction qui appellera un traitant (la fonction hand). Celui-ci été défini de telle sorte qu'il ne réagisse que lorsqu'un SIGALRM est envoyé, les autres étant bloqués.
Il contient ensuite une boucle infinie qui va bloquer le processus jusqu'à ce que le signal ait été délivré.

- timer_init

Création du thread demon, d'un sigaction qui enverra un premier SIGALRM et initialisation des 200 éléments du tableau d'échéanciers : tous les timers sont mis à 0 et expire à true.

- timer_set

Lorsque plusieurs signaux SIGALRM sont envoyés, c'est cette fonction qui va armer les temporisateurs dans le tableau d'échéanciers, notamment grâce au paramètre delay qui nous permettra de préciser le délai d'attente avant l'enclenchement de l'évènement, donc leur donner une valeur et enregistrer la valeur param passée par le second paramètre de la fonction.