Projet Programmation Système

COUVY Julien, LEYX Sebastien

6 janvier 2017

1 Introduction

Nous devions lors de ce projet, programmer à l'aide des appels systèmes vues en cours et de nos connaissance en language C, réaliser plusieurs outils que l'on peut avoir lors de la création d'un jeu. Dans un premier temps la sauvegarde et le chargement d'un jeu, puis la modification d'un niveau, ou encore l'ajout de bombe/mines (d'objets qui s'activent en un temps donné).

2 Sauvegarde et chargement des cartes

2.1 Choix

Nous avons d'abord créé une structure 'Map' composée des caractéristiques de la carte. La structure comprend les champs suivants :

```
typedef struct {
    unsigned width;
    unsigned height;
    unsigned max_obj;
    unsigned nb_objects;
} Map_s;
```

Puis nous avons créé une structure 'Objects' composée de toutes les informations que contiennent chaque pièces. La structure comprend les champs suivants :

```
typedef struct {
    int      found; // Object is found in map
    int      type;
    int      frame;
    int      solidity;
    int      destructible;
    int      collectible;
    int      generator;
    int      name_length;
    char     *name;
} Object_s;
```

Lors de la sauvegarde de la Map, on sauvegarde dans la structure map les informations grâce aux fonctions existantes.

```
    map.width      = map_width();
    map.height     = map_height();
    map.max_obj    = map_objects();
```

Puis on parcourt chaque objet de la map, que l'on sauvegarde dans une matrice. Ensuite, nous recherchons pour chaque objet présent toutes ses caractéristiques que l'on rentre dans un tableau de structure objet. Une fois tout cela stocké dans les structures, il nous suffit avec l'appel système 'write' d'écrire ces informations dans un fichier de sauvegarde binaire. On inscrit dans la structure la taille du nom de la texture (name_length) pour pouvoir lors du chargement connaitre la taille de la chaîne de caractère à récupérer.

```
ret = read(LoadFile, &objs[i].name_length,  sizeof(int));
objs[i].name = malloc(objs[i].name_length * sizeof(char));
ret = read(LoadFile, objs[i].name,          objs[i].name_length * sizeof(char));
```

Pour le chargement, il suffit de lire ce même fichier dans l'ordre que l'on avait utilisé pour le remplir, c'est-à-dire :

```
/* SaveFile Format (line breaks are for readability only, not present in binary):
    Map width  (UNSIGNED)
    Map height (UNSIGNED)
    Maximum amount of object (UNSIGNED)
    Number of objects in map & editor (UNSIGNED) -- equals to max when using set_objects.

    Object 1:
        Found physically in the map (INT)
        Type (INT)
        Frames of its decal (INT)
        Solidity (INT)
        Destructibiliy (INT)
        Collectibility (INT)
        Generator (INT)
        Length of its texture name (INT)
        Texture name (CHAR *)

    (...) Object n

    2D Matrix of the map (INT) */
```

Nous devions ensuite pouvoir modifier la taille de la map, ajouter des objets, modifier ses caractéristiques ou encore supprimer ceux qui ne sont pas présents sur la carte. Pour le traitement des options, nous avons utilisé 'getopt' :

```
struct option long_option[] =
{
            {"getwidth",    no_argument,        0,   GET_WIDTH},
            {"getheight",   no_argument,        0,   GET_HEIGHT},
            {"getobjects",  no_argument,        0,   GET_OBJECTS},
            {"getinfo",     no_argument,        0,   GET_INFO},
            {"setwidth",    required_argument,  0,   SET_WIDTH},
            {"setheight",   required_argument,  0,   SET_HEIGHT},
            {"setobjects",  required_argument,  0,   SET_OBJECTS},
            {"pruneobjects",no_argument,        0,   PRUNE_OBJ},
            {0,             0,                   0,   HELP }
};
```

Ensuite pour agrandir la taille, il suffit juste de sauvegarder au moyen de l'appel système 'read' la matrice de la map, créer une nouvelle matrice initialisée à -1 et de placer au bon endroit dans cette matrice l'ancienne. Avec un 'ftruncate' on efface l'ancienne matrice du fichier binaire, et avec 'write' on réécrit la bonne.

Fonctions associées :

```
int get_width_value(int fd)
// retourne la valeur de la largeur de la map.
int get_height_value(int fd)
// retourne la valeur de la hauteur de la map.
void go_to_matrix(int fd, int width, int height)
// Place l'offset à la position de début de la matrice.
void copy_struct_objects(Object_s *src, Object_s *dest, int i_src, int i_dst)
// Copie la structure src à partir de l'index i_src dans la structure dest à partir de l'index i_d
Object_s *read_savefile_objects(int SaveFile, int nb_objs)
// Retourne le tableau d'objets lu dans le fichier de sauvegarde.
void write_savefile_objects(int SaveFile, Object_s *objs, int i)
// Ecris dans le fichier de sauvegarde l'objet i et ses caractéristiques.
```

Pour l'ajout d'objet, il suffit de regarder grâce au champ 'found' de la structure objet si l'objet est ou non-présent dans la map. S'il ne l'est pas, on l'ajoute, s'il y est, on regarde si les caractéristiques sont les mêmes et on les remplace par les nouvelles. Fonction de recherche d'un objet dans le fichier de sauvegarde :

```
bool object_in_savefile(int SaveFile, int index, char *args[])
{

  ...

        while (ret>0)
          {
            int in_SaveFile = lseek(SaveFile, 0, SEEK_CUR);
            ret = read(SaveFile, &res, sizeof(int));
            if(ret >0)
            {
                lseek(SaveFile,6 * sizeof(int), SEEK_CUR);
                read(SaveFile, &name_length, sizeof(int));
                char *sprite = malloc(name_length * sizeof(char));
                read(SaveFile, sprite, name_length * sizeof(char));
                if (strcmp(args[index], sprite) == 0)
                  {
                        lseek(SaveFile, in_SaveFile, SEEK_SET);
                        return true;
                  }
            }
          }
        return false;
}
```

Quant à la suppression des objets non-présents, on supprime du fichier binaire tous les objets avec un found à 'false'.

Pour toutes les fonctions set du fichier 'maputils.c', nous ne l'avons pas précisé, mais, nous mettons à jour quand c'est nécessaire (modification taille de map ...) les informations de la map (width et height de la structure ...).

2.2 Résultats

— Sauvegarde et chargement des cartes

- ☑ Ecriture structure map
- ☑ Ecriture tableau de structure d'objets
- ☑ Ecriture matrice d'objets
- ☑ Lecture structure map

- ☑ Lecture tableau de structure d'objets
- ☑ Lecture matrice d'objets

— Utilitaire de manipulation de carte

- ☑ Lecture et affichage d'information de la map (Getwidth, getheight, getobjects, getinfo)
- ☑ Modification de la taille de la map (setwidth, setheight)
- ☑ Ajout ou modification d'objets (setobject)
- ☑ Suppression d'objets non présents dans la map (pruneobjects)
- ☒ Rapport sans faute de valgrind (SDL ?)

3 Gestion des temporisateurs

3.1 Choix

Nous avons d'abord créé une structure 'Event' représentant une nouvelle occurrence d'un timer (bombe ou mine). La structure comprend les champs suivants :

```
typedef struct event_s {
    unsigned long daytime; // Heure au moment où la structure est allouée
    struct itimerval delay; // Structure de temps spécifiant le délai avant l'action
    void* event_param; // Sauvegarde du paramètre passé par SDL
    struct event_s *prev; // Chainage double (élément précédent)
    struct event_s *next; // Chainage double (élément suivant)
} Event;
```

Les évènements sont ajoutés dans une liste double chaînée dès leur création (placement d'une bombe ou d'une mine par l'utilisateur) puis sont ensuite triés dans l'ordre de déclenchement. Pour se faire, on vérifie pour chaque élément de la liste si l'horaire stocké dans la structure de l'évènement ajouté au délai avant l'explosion est inférieur à ce même calcul pour les autres Events. Il est important de noter que le délai doit être converti en millisecondes si l'on se réfère au prototype de la fonction timer_set.

Fonctions associées :

```
unsigned long delay_of_event(Event *e);
bool compare_delay(Event* a, Event* b);
// Applique un tri bulles sur une liste dont on donne l'adresse du premier élément.
void sort_events(Event** head);
// Ajoute l'élément pointé par new_event dans une liste DC
void add_event(Event** head, Event** new_event);
```

Pour gérer la réception des signaux, nous créons un thread démon chargé de boucler indéfiniment en attendant les SIGALRM envoyés par le système lorsque le délai de chacun des Events atteint 0 (grâce à la fonction **setitimer** appelé dans **timer_set** dont le comportement est analogue à alarm() en plus de la gestion des structures de temps **itimerval**).

Fonctions associées :

```
void* daemon_handler(void* argp);
void signal_handler(int signo);
```

À la réception d'un SIGALRM, le thread démon se charge de l'appel à la fonction **sdl_push_event** tout en lui passant en argument le paramètre de l'évènement déclenché. De plus, il doit se charger, d'une part de libérer l'espace mémoire dans la liste de l'évènement terminé et de mettre à jour la tête de file, d'autre part de la gestion des évènements à venir. On distingue deux possibilités où :

— Un ou plusieurs évènements seraient très rapprochés dans le temps de celui venant de se déclencher. On considère le temps minimum espaçant deux évènements de façon arbitraire. Le démon doit alors répéter les étapes précédentes en négligeant la différence de temps.

```
do {
// gestion de l'Event
} while  ((curr_timer + 100000UL > next_timer) && head != NULL);
// 100000UL = 100ms, le délai minimum entre deux évènements.
```

— Les évènements suivants se déclenche avec plus de 100ms d'intervalle. Dans ce cas, il est nécessaire de mettre en place un nouveau signal d'alarme avec le délai mis à jour correspondant au prochain évènement. Cela se traduit par un nouvel appel à la fonction **setitimer** avec pour paramètre une structure **itimerval** dont le délai correspond à la différence entre le délai stocké dans la structure du prochain évènement et l'horaire au moment de l'appel.

Pour protéger les accès espaces de données partagées, nous avons mis en place un mutex que l'on initialise dès l'appel à la fonction **timer_init**. Les protections sont ajoutées dès qu'une manipulation de la liste DC à lieu notamment :

À la création d'un nouveau timer

```
void timer_set (Uint32 delay, void *param)
{
//...
    pthread_mutex_lock(&mutex);
    add_event(&head, &e);
    sort_events(&head);
//...
    pthread_mutex_unlock(&mutex);
}
```

Au sein du thread démon lors de la mise à jour de la liste

```
void* daemon_handler(void* argp)
{
//...
 while (1)
    {
        //...
        do {
            pthread_mutex_lock(&mutex);
        // SDL + mise à jour liste DC
            pthread_mutex_unlock(&mutex);
        } while ((curr_timer + 100000UL > next_timer) && head != NULL);
}
```

3.2 Résultats

— Gestion des structures de données

- ☑ Liste DC fonctionnelle
- ☑ Structure évènement
- ☑ Tri de la liste
- ☒ Rapport sans faute de valgrind (SDL ?)

— Implémentation simple

- ☑ Placement d'une bombe (timing, sprite : ok)
- ☑ Placement d'une mine (timing, armement, sprite : ok)
- ☑ Explosions respectant l'ordre chronologique

— Implémentation complète
  - ☑ Placement de plusieurs bombes
  - ☑ Placement de plusieurs mines
  - ☑ Explosions respectant l'ordre chronologique
  - ☒ Explosions simultannées (peut glitcher par moments)