Projet Programmation Système

MENAN Jimmy, PATHE Chloé

6 Janvier 2017

I. Introduction

Le but de ce projet est de développer des outils de bases à destination d'un jeu de plateforme en 2D. Le projet est divisé en deux parties, la première consistant en la manipulation et la sauvegarde est cartes créées et la deuxième consistant en la création de temporisateurs permettant de déclencher certains évènement après un court délai.

II. Partie 1 : Sauvegarde et chargements des cartes

A. Sauvegarde

Cette option est entièrement fonctionnelle. La fonction s'en occupant est ainsi capable d'ouvrir et/ou créer un fichier dont le nom est défini par le paramètre char **filename*, ainsi que d'enregistrer les informations nécessaires à la recréation de la carte par la fonction de chargement.

Ainsi, une fois le fichier ouvert en mode lecture seule, la fonction mémorise en premier lieu les informations générales de la carte. C'est à dire largeur, hauteur et nombre d'objets.

```
int fdmap = open(filename, O_WRONLY | O_TRUNC |O_CREAT, 0640);
int width = map_width();
int height = map_height();
int nb_obj = map_objects();
write(fdmap, &width, sizeof(int));
write(fdmap, &height, sizeof(int));
write(fdmap, &nb_obj, sizeof(int));
```

Ensuite, la fonction s'occupe parcourir chaque case du niveau afin de l'écrire dans le fichier.

```
for(int y = 0; y < height; ++y){
  for(int x = 0; x < width; ++x){
    obj = map_get(x, y);
    write(fdmap, &obj, sizeof(obj));
  }
}
```

Et, enfin, on enregistre l'ensemble des objets du niveau dans le fichier. Les informations sont mémorisées dans l'ordre suivant : Taille du nom de l'objet, nom de l'objet, nombre d'images, solidité, destructible, collectable, générateur.

```
for(int i = 0; i < nb_obj; ++i){
  char *name_obj = map_get_name(i);
  int name_len = 0;
  while(name_obj[name_len] != '\0'){
    name_len++;
  }
  name_len++;
  nb_frames = map_get_frames(i);
  solidity = map_get_solidity(i);
  is_destruct = map_is_destructible(i) * MAP_OBJECT_DESTRUCTIBLE;
  is_collect = map_is_collectible(i) * MAP_OBJECT_COLLECTIBLE;
  is_gen = map_is_generator(i) * MAP_OBJECT_GENERATOR;

  write(fdmap, &name_len, sizeof(int));
  write(fdmap, name_obj, (name_len*sizeof(char)));
  write(fdmap, &nb_frames, sizeof(int));
  write(fdmap, &solidity, sizeof(int));
  write(fdmap, &is_destruct, sizeof(int));
  write(fdmap, &is_collect, sizeof(int));
  write(fdmap, &is_gen, sizeof(int));
}

close(fdmap);
printf("Map successfully saved at: %s.\n", filename);
```

Deux choix ont été faits dans cette étape pour faciliter la lecture du fichier au chargement. Le premier étant de mémoriser la taille de la chaîne de caractère du nom juste avant ce dernier pour faciliter sa récupération. Le second étant de ne pas mémoriser les trois booléens sous la forme booléenne renvoyée par leurs fonctions mais sous leurs véritables valeurs.

B. Chargement

Cette option est entièrement fonctionnelle. La fonction est ainsi capable d'ouvrir un fichier *map* précédemment sauvegardée et d'en extraire les informations afin de reconstituer le niveau. Son fonctionnement est nécessairement similaire à la fonction de sauvegarde

Elle récupère en premier lieu les informations générales de la carte (largeur, hauteur, nombre d'objets) afin de calculer le nombre de données qu'elle doit récupérer par la suite.

```
int width, height, nb_obj, nb_frames, obj, name_len, solidity, destruct, collect, gen;

int fdmap = open(filename, O_RDONLY);
if(fdmap==-1){
  exit_with_error("Impossible d'ouvrir le fichier\n");
}

read(fdmap, &width, sizeof(int));
read(fdmap, &height, sizeof(int));
read(fdmap, &nb obj, sizeof(int));
```

Elle crée ensuite le niveau avec la fonction *map_allocate* puis définit chacune des cases.

```
map_allocate(width, height);
//On définit chaque case
for(int y = 0; y < height ; ++y){
  for(int x = 0; x < width; ++x){
    read(fdmap, &obj, sizeof(int));
    map_set(x, y, obj);
  }
}
```

En dernier lieu elle définit les objets, comme dans la fonction *new_map*.

```
map_object_begin(nb_obj);

//Puis on définit chaque objets.
for(int i = 0; i < nb_obj; ++i){
  read(fdmap, &name_len, sizeof(int));
  char name_obj[name_len];
  read(fdmap, name_obj, (name_len*sizeof(char)));
  read(fdmap, &nb_frames, sizeof(int));
  read(fdmap, &solidity, sizeof(int));
  read(fdmap, &destruct, sizeof(int));
  read(fdmap, &collect, sizeof(int));
  read(fdmap, &gen, sizeof(int));
  map_object_add(name_obj, nb_frames, solidity | destruct | collect | gen);
}

map_object_end();
```

C. Maputil

1. Informations élémentaires

Cette fonction est entièrement fonctionnelle. Le programme *maputil* est ainsi capable d'ouvrir un fichier map indiqué en paramètre pour y extraire et afficher les informations demandées également en paramètres. Le programme accepte ainsi en paramètre *--getwidth*, *--getheight*, *--getobjects* affichant respectivement la largeur, la hauteur ainsi le nombre d'objet d'un carte.

```
char param[15];
strcpy(param, argv[2]);
if(!strcmp(param, "--getwidth"))
  get_width(argv[1]);
else if(!strcmp(param, "--getheight"))
  get_height(argv[1]);
else if(!strcmp(param, "--getobjects"))
  get_objects(argv[1]);
else if(!strcmp(param, "--getinfo"))
  get_info(argv[1]);
else if(!strcmp(param, "--setwidth")){
  if(argc < 4){
    fprintf(stderr, "maputil <file> --setparam <size>");
    exit(1);
  }
  set_width(argv[1], atoi(argv[3]));
}
else if(!strcmp(param, "--setheight")){
  if(argc < 4){
    fprintf(stderr, "maputil <file> --setparam <size>");
    exit(1);
  }
  set_height(argv[1], atoi(argv[3]));
}
```

```
void get_width(char *filename){
  int fd = open(filename, O_RDONLY);
  if(fd==-1){
    fprintf(stderr, "Erreur ouverture du fichier\n");
    exit(1);
  }
  int width;
  read(fd, &width, sizeof(int));
  close(fd);
  printf("Map Width : %d\n", width);
}

int height;
lseek(fd, sizeof(int), SEEK_SET);
read(fd, &height, sizeof(int));

int nb_obj;
lseek(fd, sizeof(int)*2, SEEK_SET);
read(fd, &nb_obj, sizeof(int));
```

L'utilisation de *lseek* permet à la fonction de se placer là où se trouve l'information dont elle a besoin. Enfin, le paramètre *--getinfo* renvoie les trois dernières informations à la fois.

```
int w, h, o;
read(fd, &w, sizeof(int));
read(fd, &h, sizeof(int));
read(fd, &o, sizeof(int));
close(fd);
printf("Map infos : %dx%d with %d objects\n", w, h, o);
```

2. Modification de la taille de la carte

Cette fonction n'est que partiellement fonctionnelle. La fonction est actuellement capable d'ouvrir un fichier map indiqué en paramètre et d'agrandir la carte. Elle est cependant incapable de réduire la taille de celle-ci sans causer de problèmes au chargement de la map.

Cette modification se fait en faisant appel à un fichier temporaire.

```
int tmp = open("tmp.map", O_RDWR | O_CREAT);
```

Dans un premier temps, la fonction compare l'ancienne et la nouvelle taille de la carte. Si les deux sont identiques, elle n'effectue aucune modification.

```
if(old_w == new_w){
  close(fd);
  fprintf(stderr, "Same width");
  exit(1);
}
```

Dans le cas où la nouvelle taille est supérieure à l'ancienne (largeur ou hauteur), la fonction reconstitue la matrice d'entier représentant le niveau en copiant les anciennes données tout en ajoutant des cases vides (représentés par des -1) lorsqu'elle dépasse de la taille initiale.

```
if(new_w > old_w){
  for(int y = 0; y < old_h; ++y){
    for(int x = 0; x < new_w; ++x){
      if(x < old_w){
        read(fd, &buf_int, sizeof(int));
        write(tmp, &buf_int, sizeof(int));
      }
      else{
        write(tmp, &map_none, sizeof(int));
      }
    }
  }
```

Une fois la copie de la matrice terminée, elle copie également tout le reste des données (les objets) sans les traiter jusqu'à la fin du fichier.

```
char buf[1000];
int r;
while((r = read(fd, buf, 1000)) > 0){
  write(tmp, buf, r);
}
```

Avant de terminer, elle remplace alors le contenu du fichier map initial par celui du fichier temporaire avant de supprimer ce dernier.

```
char buf[1000];
int r;
int data;
lseek(src, 0, SEEK_SET);
lseek(dst, 0, SEEK_SET);
while((r = read(src, buf, 1000)) > 0){
  write(dst, buf, r);
  data+=r;
}
ftruncate(dst, data);
```

La fonction n'est cependant pas capable de réduire la taille d'une carte.

3. Remplacement des objets d'une carte

Cette fonction n'est pas implémentée. Pour fonctionner, il aurait fallu dans un premier temps compter le nombre d'objets présents en paramètres ((argc - 2) / 6) pour remplacer l'ancien nombre. Puis placer le file *descriptor* après la matrice des cases afin d'écrire les nouveaux objets en lieu et place de ceux déjà présents. Pour terminer, l'utilisation de *ftruncate* permet de raccourcir le fichier en cas de besoin.

4. Suppression des objets inutilisés

Cette fonction n'est pas implémentée. Pour fonctionner, il aurait fallu rechercher dans la matrice une occurrence de chaque objet et mémoriser le résultat. On écrirait alors dans un fichier temporaire la liste description de tous les objets que l'on garde afin de remplacer l'actuelle. Cela implique cependant de modifier le nombre d'objet inscrit au début du fichier ainsi que les indices de chaque objets. Il faudra alors parcourir à nouveau la matrice afin d'appliquer ces modifications.

D. Conclusion de la première partie

J'ai rencontré énormément de problèmes lors de projet de programmation système. Ayant accumulé beaucoup de lacunes au cours du semestre, je n'ai pas réussi à effectuer une partie du travail demandé.

En ce qui concerne la partie effectuée, je constate avec un peu de recul que je pourrais y apporter beaucoup d'améliorations tant sur la forme que sur le fond. Certaines parties du codes gagneraient en clarté à être factorisées en les implémentations dans des fonctions à part. De plus, il aurait également été possible de simplifier grandement le format du fichier de sauvegarde en ne notant que les objets et leur positions X et Y plutôt que de mémoriser l'ensemble de la matrice d'entier des cases. Le programme aurait pu être capable de compléter lui-même les trous, limitant ainsi les appels systèmes de lecture et d'écriture.

III. Deuxième partie : Gestion des temporisateurs

A. Démon récepteur de signaux

La fonction *timer_init* fonctionne. L'implémentation n'ayant pas était finie, la variable de retour est restée à 0. On rappelle que cette fonction est appelée au démarrage et permet de traiter les différents signaux. Ces différents signaux permettent, par exemple, de faire clignoter le personnage, ou de faire exploser des bombes après un certain lapse de temps.

La fonction timer_init commence par créer un masque de la manière suivante :

```
sigset_t mask;
sigemptyset(&mask);
```

```
sigaddset(&mask, SIGALRM);
```

Ce masque permet de bloquer le signal *SIGALRM* pour tous les autres threads et aurait pu être créé dans le thread, mais il semble que ce soit plus lisible ainsi. Ce signal bloquant est, lui, initialisé en faisant :

```
pthread_sigmask(SIG_BLOCK, &mask, NULL);
```

Un verrou est ensuite appliquer au thread, à des fins de protection.

```
pthread_mutex_init(&mutex, NULL);
```

Le thread est ensuite créé. Si la création a rencontré une erreur, la fonction retourne une erreur et s'arrête.

```
pthread_t thread;
if (pthread_create(&thread, NULL, &daemon, NULL) !=0){
      perror("thread init error");
      exit(EXIT_FAILURE);
}
```

Pour le thread démon en lui-même, il doit faire une boucle infinie qui appelle *SIGSUSPEND*. Le thread se compose de la manière suivante :

```
void *daemon(void *param){

 struct sigaction s;
 s.sa_handler = sig_handler;
 sigemptyset(&s.sa_mask);
 sigaddset(&s.sa_mask, SIGALRM);
 sigaction(SIGALRM, &s, NULL);

 sigset_t mask;
 sigfillset(&mask);
 sigdelset(&mask, SIGALRM);
 s.sa_flags = 0;

 while (1){
       sigsuspend(&mask);
 }
    }
```

Ne manque que le traitant du signal, qui utilise ici la fonction *thread_self*, qui indique quel thread est en cours d'exécution.

```
void sig_handler(int sig){
  printf("Thread appelé n°%d\n",(int) pthread_self());
}
```

B. Implémentation simple

Le but ici est d'implémenter un premier temporisateur pour armer les différents évènements (l'effet à retardement des bombes, etc…). La fonction *timer_set* fonctionne dans ce cas-là.

L'évènement est défini par la structure *event*.

```
typedef struct event{
 void *param; // l'evenement
 struct itimerval time; // le delai avant action en µs
 struct event *next; // l'évènement suivant
 struct event *previous; // l'évènement précédent
} event;

struct event *first = NULL;
```

Ici, *param* est l'évènement et time est le délai de déclanchement.

On choisit tout de suite de mettre dans la structure les évènements qui précèdent et suivent l'évènement manipulé, en prévision de l'implémentation plus complexe qui suit. On crée tout de suite le premier évènement, qui est nul.

La fonction *timer_set* a pour paramètre *delay*, qui est le délai avant le déclanchement de l'action *param*, aussi passé en paramètre. Cependant, le format de ce délai n'est pas correct, on commence donc par rectifier cela. La fonction *setitimer* ayant besoin à la fois du temps en secondes et en microsecondes, on implémente tout de suite deux variables.

```
unsigned long delay_us = (unsigned long) delay * 1000;
unsigned long delay_s = (unsigned long) delay%1000 * 1000;
```

On crée ensuite un nouvel évènement *event*, construit de la façon suivante :

```
struct event *new_event = malloc(sizeof(event));
new_event->param = param;
new_event->time.it_value.tv_sec = delay_s;
new_event->time.it_value.tv_usec = delay_us;
new_event->time.it_interval.tv_sec = 0;
new_event->time.it_interval.tv_usec = 0;
```

On finit enfin par appeler *setitimer* :

```
setitimer(ITIMER_REAL, &new_event->time, NULL);
```

La variable *ITIMER_REAL* est utilisée car on veut que le timer décroisse en temps réel et qu'un *SIGALRM* soit émis à la fin du délai.

La dernière chose à modifier est l'envoi du *SIGALRM*, géré par le thread démon. Pour cela, on modifie seulement le traitant.

```
void sig_handler(int sig){
  printf("sdl_push_event(%p) appelée au temps %ld\n", *param, get_time ());
}
```

C. Implémentation complète

L'implémentation complète n'a pas été finie, même si une partie des fonctions nécessaire à sa réalisation ont été implémentée.

L'implémentation complète nécessitant plusieurs évènements, ainsi que la possibilité d'en ajouter et d'en retirer, les fonctions suivantes sont nécessaires.

La première est la fonction *next_event*, qui permet d'ajouter un nouvel évènement en fonction du premier de la chaine.

La deuxième est la fonction *suppr_event*, qui supprimer un évènement de la chaine. Si le premier évènement de la chaine est vide mais que la fonction est appelée quand même, un EXIT_FAILURE est effectué.

```
void suppr_event(event* old_event){
 if (first  == NULL){
        perror("Impossible: empty row");
    exit(EXIT_FAILURE);
}
```

Il faut free les variables temporaires créées dans ces deux fonctions pour ne pas avoir de fuites mémoire.

D'autres fonctions seraient nécessaires, comme par exemple trouver quel est l'évènement qui a le délai le moins long, ou encore, une fonction de tri en fonction du délai. Cette dernière fonction entrainerait une modification de la fonction de suppression.

La partie de test n'a pas été implémentée.

D. Mise en service dans le jeu

Cette partie n'est pas été traitée.

E. Conclusion de la deuxième partie

J'ai rencontrée quelques difficultés, surtout à cause des lacunes accumulées au cours du semestre. Ces lacunes m'ont donc obligée à faire plus de recherche quant à l'implémentation de certaines structures, même celles données dans le sujet. Cela a cependant été très instructif.

Je regrette cependant ne pas avoir pu aller plus loin dans la partie trois, étant donné que je l'avais commencée.

IV. Conclusion du projet

En conclusion, nous avons tous les deux rencontré des difficultés au cours de ce projet, venant surtout de nos propres lacunes, mais aussi de notre gestion du temps et de notre vision globale de notre code. Ces points seront à améliorer.