# Rapport du Projet de Programmation Système

Halnaut Adrien, Ordonez Romain

5 janvier 2017

**Résumé**

Ce rapport retrace la réalisation du projet de programmation système sur le jeu « Mapio ». Dans ce rapport nous aborderons deux points essentiels du projet, une première partie sur la réalisation de la commande « maputil »permettant de sauvegarder, charger et/ou modifier cette sauvegarde, et dans une deuxième partie l'ajout de temporisateur afin de créer de nouveaux objets et intéractions utilisant ce mécanisme.

# Table des matières

# Introduction

Les objectifs de ce projet se déroulent en deux parties. Dans un premier temps, ils consistent à développer des mécanismes déstinés à l'utilisation d'un jeu de plateforme 2D. Il s'agit de d'implémenter des mécanismes permettant de sauvegarder et de charger une carte utilisée par le jeu, mais aussi de modifier à l'aide d'une commande la sauvegarde.

Enfin, dans la deuxième partie, il est demandé de rajouter des temporisateurs afin d'ajouter des éléments de jeu comme des bombes à retardement, des mines ou encore des objets clignotants.

Vous pouvez retrouver notre projet à l'adresse `https://github.com/Apodeus/mapio`

# Partie 1

# Développement de la première partie - Maputil

## 1.1 Compréhension des arguments

Le traitement des arguments par le programme se fait d'une traite dans une boucle `for` et en utilisant `strcmp()`. Lorsqu'une commande est reconnue, le programme vérifie si les arguments qui la suivent sont correctes de façon idépendante par rapport à la "grande" boucle `for`, ou interrompent le programme sinon. Ce genre de traitement permet d'avoir plusieurs commandes en même temps pour une seule exécution du programme (on pourra utiliser `--setheight` et `--setwidth` en même temps, par exemple). Le traitement est donc séparé en deux parties (à l'exception de `--setobjects`, plus complexe et demandant à être exécutée d'une seule traite), la première consiste à vérifier la cohérence des arguments passés au programme, puis à exécuter les ordres souhaités.

## 1.2 Commandes

Comme il est dit précedemment, la première partie du projet consistait donc à implémenter des mécanismes de sauvegarde et de chargement de carte au jeu à l'aide d'une commande « maputil ». Pour cela, il nous a tout d'abord fallu créer un format de fichier de sauvegarde afin de s'organiser pour la suite. Ainsi nous avons décidé d'organiser notre fichier de la manière suivante :

```
<largeur> <hauteur>
<nombre d'objets sur la carte>
<x> <y> <ID> pour chaque objet
...
<nombre d'objet différents>
<longueur du chemin> <chemin de l'image> <nombre de frames> <solide> <destructible>
     <collectable> <generateur> pour chaque propriété
...
```

Des commandes de base nous ont été fournis afin de récuperer des informations essentielles aux éléments présents dans le jeu, par exemple pouvoir récuperer les propriétés d'un object, savoir quel élément se situe à telles coordonnées, ou encore la taille de la carte.

Nous avons décidé d'avoir un schéma assez simple pour charger les informations depuis un fichier de sauvegarde donné, tout d'abord, nous chargeons le fichier dans une structure contenant deux listes chainées, puis nous effectuons les traitements souhaités sur cette structure, et une fois toutes les demandes de traitements terminées, nous générons un nouveau fichier de sauvegarde basé sur cette structure.

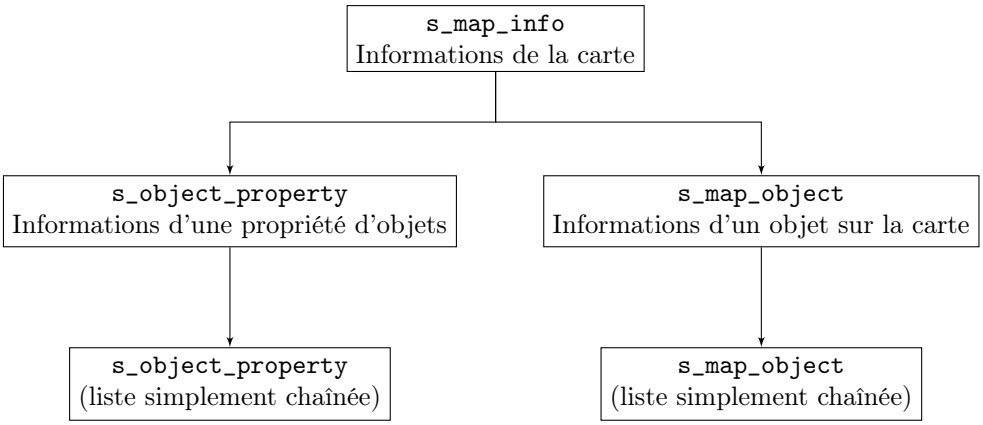


FIGURE 1.1 – Structure d'une carte dans `maputil`

Code 1.1 – maputil.c structure de la carte

```
struct s_map_info
{
   char* filename;
   int width;
   int height;
   int nb_element;
   int nb_object;

   struct s_map_object* first_case;
   struct s_object_property* first_property;
};
```

Code 1.2 – maputil.c structure des objets présents sur la carte

```
struct s_map_object
{
   int x;
   int y;
   int type;
   int active;

   struct s_map_object* next;
};
```

Code 1.3 – maputil.c structure des propriétés des objets

```
struct s_object_property
{
   int lenght_path;
   char* path;
   int frames;
   int solidity;
   int destructible;
   int collectible;
   int generator;
   int active;

   struct s_object_property* next;
};
```

Aussi, pour simplifier le traitement du fichier de sauvegarde et récuperer les informations plus facilement, nous avons utilisé la fonction `strtok()` de la librairie `string.h`, l'utilisation de cette fonction est assez simple, elle prend en paramètre une chaîne de caractère et un motif, et transforme cette chaîne en plusieurs mots normalement séparés par ce motif. Voici un exemple de son utilisation :

Code 1.4 – maputil.c

```
   int buff_size = 256;
   char buffer[buff_size];
   char* token;
   char delim[2] = " ";

   //On recupere la largeur et la hauteur
   fgets(buffer, buff_size, file_save);
    //On initialise le premier token
   token = strtok(buffer, delim);
   loading_map->width = atoi(token);
   //puis on passe au suivant en mettant NULL comme premier parametre
   token = strtok(NULL, delim);
   loading_map->height = atoi(token);
```

Nous avons tout d'abord travaillé sur les fonctions de sauvegarde et de chargement à partir d'un fichier, puis sur une option `--getinfo` permettant d'afficher les informations normalement fournis par les options `--getwidth` , `--getheight` et `--getobjects` , affichant respectivement la largeur, la hauteur, et les objets présents sur la carte.

La taille de la carte doit être modifiable à l'aide des commandes `--setwidth <w>` et `--setheight <h>`. Cette modification demande donc de changer deux valeurs, mais aussi d'enlever les objets qui ne sont plus sur la nouvelle surface de jeu, pour faciliter le traitement des objets présents ou non présents, nous avons ajouté un champ `active` à la structure des éléments (`s_map_object`) afin de les recopier ou non dans le nouveau fichier (on passe la valeur de `active` initialement à 1 à 0, les objets ayant la valeur 0 ne seront pas recopiés dans le nouveau fichier de sauvegarde).

Il doit aussi être possible de remplacer des objets à l'aide de la commande `--setobjects`.

Pour éviter d'avoir des propriétés non utilisées lors de la création d'une carte, une commande `--pruneobjects` est ajoutée afin de supprimer les propriétés inutiles, cependant, en supprimant ces propriétés, il faut parfois réattribuer les identifiants des objets de la carte (champ `type`) afin qu'il n'y ait pas de problème lors du chargement du nouveau fichier. Ici encore, pour faciliter le traitement lors de la création du nouveau fichier, nous avons ajouter un champ `active` à la structure des propriétés qui s'utilise de la même façon que précédement.

# Partie 2

# Développement de la deuxième partie - Les temporisateurs

## 2.1 Implémentation simple

L'implémentation simple du temporisateur consistait à faire en sorte que le programme créait un thread type *daemon* à son démarrage. Un *daemon* (en référence au démon de Maxwell) et un thread qui fonctionne *seul* en arrière-plan, en opposition à son utilisation permettant de faire des opérations en parallèle pour gagner en temps d'exécution, et qui intéragit avec son créateurs à certains moments plus ou moins définis, aussi appelés *events*.

Le daemon demandé devait pouvoir intercepter les signaux type `SIGALRM` et l'indiquer à la console au moyen d'un `printf()`.

Le sujet demandait d'utiliser `sigsuspend()` mais nous n'avons pas réussi à faire fonctionner le thread uniquement avec cette fonction, et nous avons donc eu besoin d'avoir recours à `sigaction()`, celle-ci faisant effectuer les fonctions handlers sur le thread principal et non sur le daemon.

Le problème que nous avons eu est que le signal `SIGALRM`, par défaut, fait arrêter le programme le reçevant. Ainsi il fallait le bloquer sur le thread principal et le laisser passer sur le daemon, même en utilisant des fonctions comme `sigprocmask()` ou `pthread_sigmask()`, le signal passait sur le thread principal et celui-ci s'arrêtait donc. Un moyen que nous avons trouvé pour bloquer le signal sur le thread principal était d'utiliser `signal(SIGALRM, SIG_IGN)`, mais cette fonction faisait ignorer le signal au daemon aussi, le rendant non fonctionnel.

## 2.2 Implémentation complète

L'implémentation complète du temporisateur demandait une gestion précise des différentes alarmes émises par le jeu. Nous avons donc, encore une fois, décidé d'organiser tout celà via des structures. Nous avons donc implémenter une structure `s_event` qui reprend le principe des listes simplement chaînées :

Code 2.1 – tempo.c

```
struct s_event{

   void* param_event;
   unsigned long done_time;

   struct s_event* next;
};
```

`param_event` contient l'argument pour `sdl_push_event()` prévu par le jeu, et `done_time` le temps auquel l'alarme doit se déclencher, calculé dans `timer_set()`.

A chaque ajout de nouvelle alarme, un `s_event` est créé et est placé dans la liste en fonction de son `done_time`, afin de les garder triés dans l'ordre croissant : l'event devant arriver le plus tôt sera au début de la liste et celui arrivant le plus tard à la fin.

Code 2.2 – tempo.c

```
void add_new_event(event e)
{
   event prev_event = NULL;
   event current_event = root_event;
   while (current_event!= NULL && current_event->next != NULL && e->done_time >
        current_event->done_time)
   {
      prev_event = current_event;
      current_event = current_event->next;
   }

   if (root_event == NULL)
      root_event = e;

   if (prev_event != NULL)
      prev_event->next = e;

   if (current_event != NULL)
      e->next = current_event;
}
```

A chaque ajout d'event dans la liste, un `setitimer()` est appelé avec un interval de temps correspondant au moment présent et au temps auquel le premier event de la liste devrait arriver. Comme il ne peut y avoir qu'un seul timer à la fois, l'ancien est remplacé par le nouveau.

Code 2.3 – tempo.c

```
void handler()
{
   pthread_mutex_lock(&lock);
   unsigned long date = root_event->done_time;
   while(root_event != NULL && root_event->done_time/100 == date/100)
   {
      sdl_push_event(root_event->param_event);
      event prev_event = root_event;
      if (root_event->next != NULL)
         root_event = root_event->next;
      else
         root_event = NULL;

      free(prev_event);
   }

   pthread_mutex_unlock(&lock);
}
```

Un intervalle est aussi appliqué dans la fonction `handler()` pour vérifier si il n'y a pas d'events trops proches l'un de l'autre dans la liste, et les exécute en

même temps si c'est le cas (à la 0.1ms près) en divisant leurs temps par 100 lors de la vérification :

Code 2.4 – tempo.c

```
void timer_set (Uint32 delay, void *param)
{
   pthread_mutex_lock(&lock);
   event e = (event) malloc(sizeof(struct s_event));
   e->param_event = param;
   e->done_time = get_time() + delay * 1000;
   e->timer = delay;
   e->next = NULL;
   add_new_event(e);

   struct itimerval timer;
   timer.it_interval.tv_sec = 0;
   timer.it_interval.tv_usec = 0;
   timer.it_value.tv_sec = (root_event->done_time - get_time()) / 1000000;
   timer.it_value.tv_usec = ((root_event->done_time - get_time()) % 1000000);
   setitimer(ITIMER_REAL, &timer, NULL);
   pthread_mutex_unlock(&lock);
}
```

On remarquera dans ces deux derniers extraits de code l'utilisation de *mutex* ; Comme la liste d'event peut à tout moment être utilisée et modifiée par le `sigaction`, il faut garantir qu'un élement de la liste en cours d'utilisation ne soit pas modifié ou même effacé et libéré par l'une des deux fonctions. Les *mutex* permettent d'éviter ce genre de conflit en laissant qu'un seul thread accéder à la ressource à la fois, et en bloquant les autres.

En ce qui concerne du jeu, à part quelques rares bugs décris dans la partie suivante, les bombes et mines ont fonctionné correctement dès le passage à 1 pour le retour de `timer_init()`.

# Partie 3

# Grille d'auto-évaluation

| *Fonctionnalité* | *Difficulté* | *Score/10* |
|---|---|---|
| **Sauvegarde et chargement des cartes** | | |
| Sauvegarde | + | 10 |
| Chargement | + | 10 |
| Informations élémentaires (e.g. `maputil -getinfo`) | + | 10 |
| Modification de la taille de la carte | ++ | 10 |
| Remplacement des objets d'une carte | +++ | 10 |
| Suppression des objets inutilisés | ++ | 10 |
| **Gestion des temporisateurs** | | |
| Réception des signaux par un thread démon | + | 5 |
| Implémentation simple (un temporisateur à la fois) | + | 10 |
| Implémentation complète (et protocole de test) | +++ | 7 |
| Mise en service dans le jeu | ++ | 7 |

Des bugs concernant les temporisateurs ont été aperçus lors des tests, par exemple lorsque deux bombes touchent le personnage au même instant, le clignotement du personnage ne s'arrête pas. Aussi, mais très rarement, des bombes peuvent ne pas exploser (a été aperçu qu'une seule fois lors du développement et n'a pas réapparu, sans pour autant qu'un correctif particulier ait été appliqué pour ce cas).