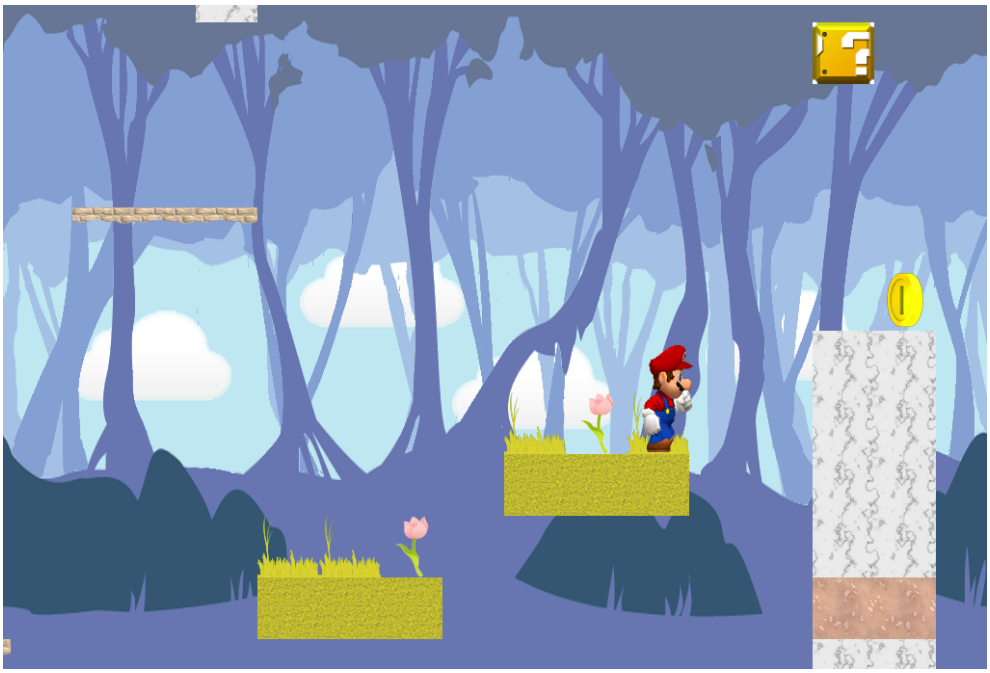
Programmation Système — Projet 2016

1 Objectifs

Il s'agit de développer quelques mécanismes de base destinés à servir dans le développement d'un petit jeu de plateforme [1] 2D. Le projet se déroulera en plusieurs parties distinctes. La première porte sur la mise en place d'un mécanisme de sauvegarde et de chargement de la carte utilisée dans le jeu.



2 Prise en main

Avant toute chose, jetons un oeil au fonctionnement du jeu. Placez vous dans le répertoire racine et tapez juste `make`. *Attention, la compilation repose sur une bibliothèque* `libgame.a` *qui a été fabriquée sur les machines du CREMI. Il n'y a pas de garantie particulière qu'elle fonctionne sur d'autres configurations Linux.*

2.1 Commandes de base

Si tout se passe bien, un programme binaire a été produit et vous pouvez lancer la commande : `./game`. Vous pouvez alors vous promener avec un personnage (flèches gauche et droite), sauter (flèche vers le haut) et... tirer des missiles (espace). La carte du jeu est plus grande que la surface affichée à l'écran : contemplez le *scrolling* du décor !

1. « À la Mario »

Bon d'accord, il faut avouer que la carte est un peu vide pour l'instant.

2.2 Editeur de niveau

Heureusement, vous pouvez éditer la carte interactivement en appuyant sur la touche `e`. À ce moment, vous voyez une texture apparaître en surbrillance en haut à gauche. Vous pouvez vous déplacer (toujours avec les flèches) pour positionner cette sorte de *tampon* où vous voulez.

Dans ce mode édition, il y a deux touches utiles :

Espace Permet d'appliquer le tampon, et donc d'ajouter un objet à la carte avec la texture courante. Si vous appliquez le tampon au-dessus d'un objet qui possède la même texture, l'objet est effacé, sinon le nouvel objet remplace l'ancien.

Tab Permet de passer à une texture différente.

Un appui sur la touche `e` (ou sur `Escape`) permet de quitter le mode édition et de revenir au jeu.

2.3 À propos des objets de la carte

Remarquez les objets n'ont pas tous la même propriété. Certains sont solides (on ne peut jamais les traverser), d'autres sont semi-solides (on peut les traverser vers le haut, mais ils se comportent comme un plancher lorsque l'on tombe dessus, d'autres encore sont comme de l'air : on peut les traverser.

Certains objets sont également destructibles (les blocs de marbre par exemple), ce qui est une propriété orthogonale à la précédente.

2.4 Initialisation de la carte

Dans l'état actuel, le programme est dépourvu de fonctions de sauvegarde/chargement de la carte : les modifications effectuées en mode édition sont perdues lorsque l'on quitte le jeu, car celui-ci redémarre toujours dans la même configuration fixe.

Pour comprendre comment est créée la carte, il faut regarder la fonction `map_new`, définie dans le fichier `src/mapio.c` :

```
1  void map_new (unsigned width, unsigned height)
2
3  map_allocate (width, height);
4
5  for (int x = 0; x < width; x++)
6    map_set (x, height - 1, 0); // Ground
7
8  for (int y = 0; y < height - 1; y++) {
9    map_set (0, y, 1); // Wall
10   map_set (width - 1, y, 1); // Wall
11 }
12
```

```
13 map_object_begin (4);
14
15 map_object_add ("images/ground.png", 1, MAP_OBJECT_SOLID);
16 map_object_add ("images/wall.png", 1, MAP_OBJECT_SOLID);
17 map_object_add ("images/grass.png", 1, MAP_OBJECT_SEMI_SOLID);
18 map_object_add ("images/marble.png", 1, MAP_OBJECT_SOLID |
19                                          MAP_OBJECT_DESTRUCTIBLE);
20
21 map_object_end ();
22 }
```

Comme vous pouvez le deviner, la fonction `map_allocate` alloue une carte en mémoire de dimension `width` $\times$ `height`. En interne, vous pouvez imaginer que la carte est juste un tableau d'entiers à deux dimensions. Chaque case du tableau représente un numéro d'objet, ou -1 (constante `MAP_OBJECT_NONE`) si la carte ne contient pas d'objet à cet endroit. Au sortir de la fonction `map_allocate`, la carte ne contient aucun objet.

Les lignes suivantes (5 à 11) servent à placer des objets pour construire un sol et des murs, à l'aide de `map_set`. La case $(0, 0)$ est celle en haut à gauche (i.e. la numérotation des lignes/colonnes suit la même convention que la numérotation des pixels dans une image). Vous pouvez remarquer que le numéro de l'objet « sol » est 0, et le numéro de l'objet « mur » est 1.

La dernière partie de la fonction (lignes 13 à 21) est responsable du chargement des caractéristiques de chaque objet de la carte. Dans cet exemple, la carte contient 4 types d'objets (comme vous avez pu le voir en jouant avec le mode édition). Pour chacun de ces objets, un appel à `map_object_add` est chargé d'indiquer :

1. le nom du fichier contenant la représentation graphique de l'objet (un fichier image contenant des *sprites* de 64×64 pixels) ;
2. le nombre de sprites contenus dans le fichier (1 pour un objet d'apparence fixe, ou un nombre supérieur pour un objet animé ;
3. un entier formé par la combinaison de plusieurs propriétés d'un objet (eg. `MAP_OBJECT_SOLID`). Les valeurs possibles pour les propriétés sont listées dans le fichier d'entête `include/map.h`.

2.5 Modification de la carte initiale

Modifiez la fonction `map_new` pour include deux nouveaux objets :

1. des fleurs, uniquement destinées à égayer le paysage (`"images/flower.png"`), qui auront donc comme propriété d'être de type `MAP_OBJECT_AIR`
2. des pièces à collecter, qui sont des objets animés (le fichier `"images/coin.png"` est composé de 20 sprites) qui ont comme propriétés `MAP_OBJECT_AIR` et `MAP_OBJECT_COLLECTIBLE`.

3 Première Partie : Sauvegarde et chargement des cartes

Il s'agit maintenant de développer les fonctions de sauvegarde (`map_save`) et de chargement (`map_load`) des cartes.

Pour l'instant, ces fonctions sont vides, mais elles sont prêtes à être utilisées dans le jeu (eg. un appui sur la touche `s` provoque l'exécution de la fonction `map_save("saved.map")` par exemple).

Vous avez une certaine liberté quant à la façon donc vous allez mémoriser dans un fichier les différentes informations relatives à une carte. Vous utiliserez pour cela l'interface des appels systèmes pour la gestion de fichiers (`open`, `read`, `write`, `lseek`, etc.) Veillez toutefois à bien lire l'intégralité des questions qui suivent avant de choisir dans quel ordre vous allez enregistrer les informations...

3.1 Sauvegarde

En observant la fonction `map_new`, on sait qu'une carte est caractérisée par :
- ses dimensions (largeur et hauteur) ;
- le nombre d'objets différents qu'elle peut contenir ;
- le contenu de chaque case de la carte (une matrice d'entiers) ;
- les caractéristiques de chaque type d'objet (nom du fichier image, nombre de sprites, propriétés).

Ces différentes informations sont accessibles au moyens des fonctions suivantes :

map_width() (resp. map_height()) retourne la largeur de la carte (resp. la hauteur) ;

map_objects() retourne le nombre d'objets différents que la carte renferme ;

map_get(x, y) retourne le type d'objet en (x, y) ou `MAP_OBJECT_NONE` ;

map_get_name(obj) retourne un pointeur sur le chemin d'accès au fichier PNG décrivant la texture de l'objet ;

map_get_frames(obj) retourne le nombre de sprites ;

map_get_solidity(obj) retourne `MAP_SOLID`, `MAP_SEMI_SOLID` ou `MAP_AIR` ;

map_is_destructible(obj) retourne 1 si l'objet est destructible, 0 sinon ;

map_is_collectible(obj) idem ;

map_is_generator(obj) idem.

N'oubliez pas que la longueur des chaînes de caractères (pour le nom des textures) n'est pas fixe, et qu'il faut penser à faciliter l'opération de chargement...

3.2 Chargement

La fonction `map_load` doit effectuer à peu près les mêmes opérations que celles effectuées dans `map_new`, à ceci près que les données doivent provenir d'un fichier.

Lors du chargement des éléments de la carte, vérifiez que le numéro de chaque objet est bien compris entre -1 et le nombre d'objets total -1.

À partir de maintenant, vous êtes en mesure de tester que le couple sauvegarde/chargement fonctionne :

1. Sauvez une carte à l'aide de la touche `s`
2. Relancez le programme de la façon suivante : `./game -l maps/saved.map`

3.3 Utilitaire de manipulation de carte

Il s'agit de développer un petit programme `maputil` capable de modifier des fichiers contenant une carte. Les sources de cet utilitaire pourront être placées dans le sous-répertoire `util`.

Cet utilitaire permettra de consulter les informations contenues dans un fichier `file`, telles que la largeur actuelle de la carte, la hauteur, le nombre d'objets contenus, etc. L'option `-getinfo` doit permettre d'afficher toutes ces informations en une commande. Voici la syntaxe pour ces fonctions :

```
maputil <file> --getwidth
maputil <file> --getheight
maputil <file> --getobjects
maputil <file> --getinfo
```

3.3.1 Modification de la taille de la carte

L'une des véritables fonctions que l'on attend de `maputil` est de pouvoir augmenter ou diminuer la taille d'une carte en largeur, en hauteur ou les deux :

```
maputil <file> --setwidth <w>
maputil <file> --setheight <h>
```

Lors de l'agrandissement de la carte, attention à ne pas perdre les objets disposés sur l'ancienne carte : il faut simplement ajouter de nouvelles colonnes (ou lignes) initialisées à `MAP_OBJECT_NONE`, en conservant les autres valeurs.

Lors du rétrécissement, on accepte bien entendu de perdre les objets situés sur la zone de carte qui n'existe plus.

Ces deux opérations s'effectuent en haut de la carte lorsque la hauteur est modifiée, et à droite lorsque la largeur est modifiée.

3.3.2 Remplacement des objets d'une carte

La seconde fonction de `maputil` est de réinitialiser la liste des objets qui composent une carte. On prendra pour contrainte que la nouvelle liste doit contenir autant ou plus d'objets qu'à l'origine. Les caractéristiques des objets sont passés en argument de la façon suivante :

```
maputil <file> --setobjects { <filename> <frames> <solidity>
<destructible> <collectible> <generator> }
```

Exemple d'utilisation :

```
./util/maputil maps/saved.map --setobjects "images/ground.png" 1 solid
not-destructible not-collectible not-generator "images/wall.png" 1
solid not-destructible not-collectible not-generator
```

Pour davantage de confort, il vaut mieux s'aider de la commande `xargs` pour spécifier plusieurs objets. Voici un exemple utilisant les objets spécifiés dans le fichier `util/objects.txt` (qui vous est fourni) :

```
xargs ./util/maputil maps/saved.map --setobjects < util/objects.txt
```

NB : Il se peut que la taille du fichier `<file>` diminue lors qu'une telle opération, on pourrra donc utiliser l'appel système `ftruncate` pour raccourcir le fichier, le cas échéant.

3.3.3 Suppression des objets inutilisés

On ajoutera une option permettant de supprimer du fichier les objets dont aucune occurence n'apparaît sur la carte :

```
maputil <file> --pruneobjects
```

4 Deuxième partie : Gestion des temporisateurs

Dans sa vesion un peu plus avancée, le jeu a besoin d'armer régulièrement des temporisateurs pour effectuer des transitions entre animations (bombes à retardement, mines, personnage qui clignote lorsqu'il se cogne, etc.) On se propose donc d'implémenter un gestionnaire de temporisateurs qui permettra au jeu de planifier les moments auxquels on voudra déclencher un *événement*.

Vous implémenterez votre gestion des temporisateurs dans le fichier `tempo.c`, en utilisant la possibilité qu'offre le système d'exploitation de vous envoyer un signal `SIGALRM` au bout d'un délai donné.

L'interface d'utilisation de ces temporisateurs est très simple et est définie dans `timer.h` :

— La fonction `timer_init` est appelée au démarrage et vous permet d'initialiser vos variables, de mettre en place vos traitants de signaux éventuels, etc. C'est aussi dans cette fonction que vous effectuerez vos tests en phase de mise au point. Pendant toute cette phase, la fonction doit retourner `0` pour indiquer que l'implémentation n'est pas encore complètement fonctionnelle. Lorsque vous serez confiant en votre implémentation, vous pourrez retourner `1`, et à ce moment le jeu débloquera du contenu supplémentaire qui nécessite l'utilisation de temporisateurs.

— La fonction `timer_set` permet d'armer un temporisateur. Le paramètre `delay` spécifie, en millisecondes, la durée de la période au bout de laquelle vous souhaitez déclencher un événement. Pour déclencher cet événement le moment venu, il suffira d'appeler une fonction prédéfinie `sdl_push_event` en lui passant la valeur `param` spécifiée en second paramètre de `timer_set`.

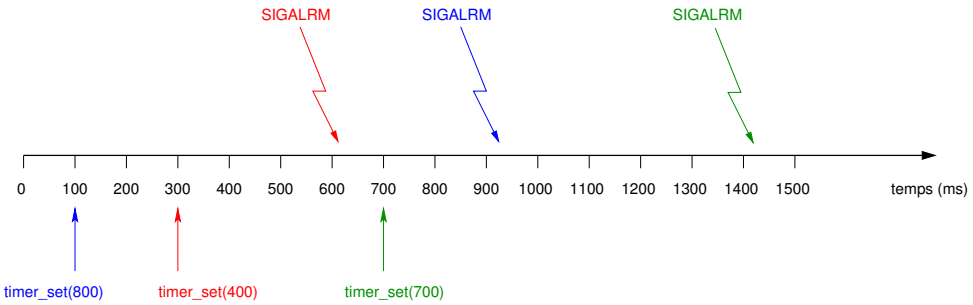


FIGURE 1 – Exemple de chronogramme

Pour fixer les idées, la figure 1 illustre le comportement attendu par votre implémentation sur un exemple simple où le programme principal arme successivement 3 temporisateurs :

— Au temps $t = 100$, le programme arme un temporisateur qui devra expirer $800ms$ plus tard, c'est-à-dire au temps $t = 900$ environ. Votre implémentation doit donc demander au noyau (au moyen de l'appel système `setitimer`) la génération d'un signal SIGALRM au bout de $800ms$.

— Au temps $t = 300$, un second temporisateur est armé. Comme son échéance est plus proche que celle du premier, on doit remplacer la première demande au noyau par une nouvelle (toujours avec `setitimer`) pour que `SIGALRM` soit délivré au temps $t = 600$.

— Aux environs de $t = 600$, le signal `SIGALRM` est rattrapé et il faut donc déclencher l'événement correspondant au second temporisateur en appelant `sdl_push_event` avec le bon paramètre. Il faut aussi re-demander au noyau la délivrance d'un `SIGALRM` dans $900 - 600 = 300ms$.

— Au temps $t = 700$, un troisième temporisateur est armé, mais son échéance est ultérieure à celle prévue pour le premier temporisateur (bleu). Il n'est donc pas nécessaire de s'adresser au noyau pour changer quoi que ce soit.

— Aux environs de $t = 900$, l'événement correspondant au premier temporisateur devra être déclenché, et il faudra un nouvel appel à `setitimer` pour demander au noyau d'envoyer un `SIGALRM` $1400 - 900 = 500ms$ plus tard...

4.1 Démon récepteur de signaux

Dans `timer_init`, créez un *thread* qui effectuera une boucle infinie en appelant `sigsuspend`. Faites en sorte que ce thread soit l'unique thread du processus qui puisse recevoir les signaux `SIGALRM`.

Testez votre implémentation en générant plusieurs signaux SIGALRM une fois le thread lancé, et en affichant l'identité du thread courant (fonction `pthread_self`) dans le traitant de signal.

4.2 Implémentation simple

On va tout d'abord considérer qu'il n'y aura qu'un seul temporisateur armé à tout instant, c'est-à-dire qu'il n'est pas nécessaire de maintenir un échéancier complet : à chaque appel de `timer_set`, on appelle simplement `setitimer` après avoir enregistré la valeur `param` quelque part. Une fois `SIGALRM` récupéré, le thread démon pourra simplement exécuter un `printf` :

```
printf ("sdl_push_event(%p) appelée au temps %ld\n", param, get_time ());
```

4.3 Implémentation complète

Étendez votre implémentation pour gérer un nombre arbitraire de temporisateurs armés n'importe quand, comme dans l'exemple de la figure 1.

Vous devrez probablement maintenir un échéancier dans lequel les jalons seront triés chronologiquement. Pensez également aux situations où certains jalons sont temporellement proches, voire simultanés : dans ce cas, la délivrance de `SIGALRM` devra traiter plusieurs événements d'une traite.

Comme votre échéancier pourra être utilisé de manière concurrente par le programme principal (dans `timer_set`) et par le thread récepteur de signaux (dans le traitant), veillez bien à protéger les accès aux structures de données partagées !

4.4 Mise en service dans le jeu

Lorsque vous serez confiant dans votre implémentation, vous pourrez remplacer le `printf` par un appel à `sdl_push_event` et retourner 1 à la fin de `timer_init`.

Si tout fonctionne correctement, vous devriez désormais pouvoir déposer des bombes (flèche vers le bas) et poser des mines (touche « t »)…