Rapport de projet : Groupe 9

Gaëlle BRAUD - Arthur LONGUEFOSSE - Gabriel LORRAIN
Université de Bordeaux
L3 Informatique
gaelle.braud@etu.u-bordeaux.fr
arthur.longuefosse@etu.u-bordeaux.fr
gabriel.lorrain@etu.u-bordeaux.fr

10 décembre 2017

Table des matières

1 Rendu intermédiaire n°1 (17 novembre)

1.1 Objectifs à atteindre

L'objectif du premier rendu était de se familiariser avec le jeu (commandes de base, édition de la carte..), puis d'inclure deux nouveaux objets (*flower* et *coin*) et leurs propriétés.
Ensuite nous devions implémenter la sauvegarde et le chargement des cartes, en développant les fonctions `map_save` et `map_load`.

Fichier à rendre :

```
mapio.c
├── map_save
└── map_load
```

1.2 Travail réalisé et résultats obtenus

Pour inclure les deux nouveaux objets, nous avons ajouter 2 au nombre d'objets total (précédemment 6) : `map_objet_begin(8)`, puis nous avons appeler un `map_object_add` pour chaque objet, en spécifiant en paramètre l'image `.png`, le nombre de sprites et les propriétés.
Les deux objets peuvent désormais être sélectionnés et affichés dans le jeu.

Pour implémenter la sauvegarde d'une carte, nous avons compléter la fonction `map_save`. Pour ce faire, nous ouvrons un fichier vide `save` avec la fonction `open` puis nous stockons chaque caractéristiques de la carte dans des variables grâce aux fonctions fournies :

- Largeur : `int width = map_width();`
- Hauteur : `int height = map_height();`
- etc...

Ensuite nous stockons de la même manière les caractéristiques des objets (en les parcourant un par un), et le contenu de chaque case en parcourant la hauteur et la largeur de la carte.
A chaque définition de variable, un appel à la fonction `write` est effectué, pour placer dans le fichier `save` les caractéristiques de la carte et des objets. Cet appel est placé dans une condition pour pouvoir gérer les erreurs :

```
int width = map_width();
if(write(save,&width,sizeof(int))!=sizeof(int)){
  exit_with_error("erreur write width");
}
```

Pour implémenter le chargement d'une carte, nous avons compléter la fonction `map_load`. Pour ce faire, nous ouvrons le fichier contenant la sauvegarde avec `open`, puis nous stockons les caractéristiques contenues dans le fichier dans des variables grâce à la fonction `read`. Chaque appel est également placé dans une condition pour pouvoir gérer les erreurs :

```
int width;
if(read(fd,&width,sizeof(int)) == -1){
   exit_with_error("erreur lecture width");
}
```

Ensuite, nous utilisons les fonctions fournies pour créer la carte :
`map_allocate(width,height)` et `map_object_begin(nb_objets)`
De la même manière que le `map_save`, nous stockons les caractéristiques des objets dans des variables en les parcourant un par un, et à chaque fin de boucle on appelle `map_object_add(nom, frame, prop)`.
Une fois tous les objets ajoutés, on termine par `map_object_end()`.
Enfin, le contenu de chaque case est stocké en parcourant la hauteur (`i de 0 à height`) et la largeur (`i de 0 à width`) de la carte, et à chaque case on appelle `map_set(j,i,contenu)` (nous avons choisi de sauver le contenu ligne par ligne, d'où l'inversion dans la boucle).

La sauvegarde et le chargement sont désormais fonctionnels :

- Pour sauvegarder une map, on appuie simplement sur la touche **s** dans le jeu, ce qui va créer ou écraser le fichier save.map contenu dans le dossier /maps.
- Pour charger une carte, il suffit d'appeler le fichier de sauvegarde du dossier /maps : `./game -l maps/saved.map`

1.3 Difficultés rencontrées

- SAVE : Propriété get solidity en binaire

La principale difficulté rencontrée pour le `map_load` était de récupérer le nom des objets (pour afficher l'image) ; nous avions pensé à utiliser un `malloc` pour initialiser le tableau contenant les caractères du nom, mais le tableau de caractère ne se remplissait pas correctement. La solution était d'utiliser un `calloc` pour bien initialiser les éléments du tableau à 0.

2 Rendu intermédiaire n°2 (30 novembre)

2.1 Objectifs à atteindre

L'objectif du deuxième rendu était de développer des utilitaires de manipulation de carte via le programme `maputil`.
Celui-ci doit permettre :

- l'affichage des informations du fichier (`getinfo`) ;
- la modification de la taille de la carte (`setwidth, setheight`) ;
- le remplacement des objets d'une carte (`setobjects`) ;
- la suppression des objets inutilisés (`pruneobjects`).

Fichiers à rendre :

```
util
└── maputil.c
```

2.2 Travail réalisé et résultats obtenus

La fonction `getInfo` correspond simplement à l'appel de trois fonctions :

- `getWidth` (largeur) ;
- `getHeight` (hauteur) ;
- `getObjects` (nombre d'objets).

Ces trois fonctions sont simplement un `read` d'un `int` sur le fichier de sauvegarde. Le choix de la valeur(`int`) à lire se fait grâce à `lseek`, qui va permettre de déplacer la tête de lecture sur le fichier de sauvegarde :

int	**déplacement à effectuer**
`width`	0
`height`	`lseek(fd,sizeof(int),SEEK_SET)`
`nb_objects`	`lseek(fd,2*sizeof(int),SEEK_SET)`

Nous avons pensé plus tard à ajouter dans le getObjects le nom et les propriétés des différents objets en plus du nombre d'objets de la carte. Nous avons simplement rajouté une boucle de taille `nombre d'objets` lisant et affichant pour chaque objet son nom (boucle d'affichage de chaque caractère de taille définie par le premier int qui correspond à la taille du nom), le nombre de frames et ses différentes propriétés (on récupère le int prop et chaque puissance de 2 nous donne une propriété, par exemple la 8ème nous donne 1 s'il est générateur, et nous affichons "générateur" etc.).
Pour réaliser le reste des fonctions de `maputil`, nous avons fait le choix de réécrire entièrement une nouvelle carte (`new.map`) avec les attributs à changer

(`width, height`, objets à remplacer ou suppression des objets inutilisés) et de remplacer l'ancienne carte par la nouvelle, définie de la manière suivante :

Pour `setWidth` et `setHeight`, on récupère chaque caractéristique de la carte avec un `read`, et on les place dans la nouvelle carte avec un `write` avec le nouvel attribut (`width` ou `height`). La boucle implémentant le contenu de chaque cases va également s'adapter avec le nouvel attribut, en supprimant le contenu des cases si il diminue, ou en ajoutant des cases vides si il augmente.

Pour `setObject`, ce sont les objets qui vont désormais être modifiés. On récupère donc les caractéristiques (inchangées) de la carte dans la nouvelle carte, puis à partir des arguments on récupère les propriétés des objets. Pour cela, nous avons implémenter une boucle `for` qui va s'incrémenter de 6 à chaque fin de boucle, pour permettre de récupérer les 6 caractéristiques de chaque objet dans la boucle. (`nom, frame, solidity, is_destructible, is_collectible ,is_generator`).

Enfin, pour `pruneObject`, on récupère les caractéristiques (inchangées) de la carte dans la nouvelle carte, puis on supprime les objets non présents sur la carte. Pour cela, on initialise un tableau `occ[nb_objet]`, puis on parcourt toutes les cases de la carte : dès qu'objet est trouvé, on incrémente la case du tableau correspondant à l'objet. Les objets non présents sur la carte (`i.e. occ[ob] = 0`) ne seront pas défini sur la nouvelle carte, grâce à un `lseek` sur leurs propriétés.

L'utilitaire de manipulation de carte `maputil` est désormais fonctionnel, depuis le dossier `util` :

— Obtenir des informations sur la carte sauvegardée :

```
./maputil ../maps/saved.map --getwidth
./maputil ../maps/saved.map --getheight
./maputil ../maps/saved.map --getobjects
./maputil ../maps/saved.map --getinfo
```

— Changer la taille de la carte :

```
./maputil ../maps/saved.map --setwidth <w>
./maputil ../maps/saved.map --setheight <h>
```

— Remplacer des objets de la carte :

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

— Supprimer les objets qui n'apparaissent pas sur la carte :

```
./maputil ../maps/saved.map --pruneobjects
```

2.3 Difficultés rencontrées

La principale difficulté rencontrée pour réaliser `maputil` était de savoir quelle méthode utiliser pour modifier les propriétés de la carte et des objets. Nous avons opté pour une redéfinition entière d'une nouvelle carte, car nous pouvons modifier toutes les propriétés à la création. Il nous suffisait ensuite de remplacer l'ancienne carte par la nouvelle.

Nous avons remarqué après le rendu qu'il y avait une erreur dans le setObjects. En effet, dans l'écriture des propriétés, en l'occurence collectible (ainsi que destructible et generator), nous avons mis MAP_OBJECT_COLLECTIBLE (qui vaut 128) au lieu de 1, donc l'écriture des propriétés était erronée. Nous avons changé ces trois propriétés, actuellement le setObjects est entièrement fonctionnel.

3 Rendu final (10 décembre)

3.1 Objectifs à atteindre

L'objectif du rendu final était d'implémenter un gestionnaire de temporisateurs, qui permettra au jeu de planifier des événements.
Pour cela, nous devions compléter les fonctions :

- `timer_init`, qui permet l'initialisation des variables et la mise en place des traitants de signaux ;
- `timer_set`, qui permet *d'armer* un temporisateur, grâce au paramètre `delay` qui spécifie la durée avant qu'un événement ne se déclenche ;

- **Bonus** : `timer_cancel`, qui permet d'annuler un temporisateur précédemment armé avec `timer_set`.

Fichiers à rendre :

- `tempo.c`
  - `map_save`
  - `map_load`
- `rapport PDF`

3.2 Travail réalisé et résultats obtenus

Tout d'abord, nous avons considéré qu'un seul évènement était déclenché à la fois.

Nous avons commencé par implémenter `timer_init` qui crée un thread et une sigaction.

La sigaction lance le traitant à la réception du signal SIGALRM seulement.

Afin que le thread soit le seul à recevoir le signal SIGALRM, nous avons masqué tous les signaux grâce à un `sigfillset` dans cette fonction `timer_init`, puis nous avons démasqué SIGALRM grâce à `sigdelset` dans la routine du thread (fonction `void * routine`) lancée en continu à la création de celui-ci avec `pthread_create`. Cette routine effectue une boucle infinie attendant la réception d'un SIGALRM avec la fonction `sigsuspend`.

Nous avons ensuite pu créer un timer dans `timer_set` qui déclenche un SIGALRM au bout d'un certain temps (delay spécifié dans les paramètres de la fonction). Afin de sauvegarder le paramètre param de la fonction, nous avons créé une variable globale `function` pour que le traitant de la sigaction y ait accès.

Le traitant de signal n'a plus qu'à déclencher l'évènement correspondant

au paramètre (param) sauvé dans function, grâce à la fonction `sdl_push_event`.

Cette implémentation nous a permis de gérer un évènement, mais gérait seulement le dernier évènement si nous en lancions plusieurs sans attendre leurs fins, puisque chaque nouvel évènement écrasait le `delay` et `param` du précédent.

Une solution possible est de stocker dans une liste les différents évènements. Nous avons décidé d'y stocker l'`heure d'appel` de l'évènement, son `délai initial`, son `délai effectif` (délai restant à un temps t) et son `paramètre` (pour l'appel à la fonction sdl_push_event). L'heure d'appel nous permet de calculer le délai restant à un temps ultérieur. Ainsi, lors de la demande d'un nouvel évènement, deux cas de figure se présentent.

Soit il n'y a aucun évènement en cours, dans quel cas nous ajoutons le nouveau en tête de liste et lançons le timer (nous l'avons mis dans une fonction `timer_launcher` pour plus de clarté).

Si un ou plusieurs évènements sont en cours, nous pouvons comparer le délai restant de l'évènement lancé avec ceux des différents éléments en cours, et placer celui-ci dans la liste (grâce à la fonction `add_timer`), triée dans l'ordre croissant des délais effectifs.

Le rôle du traitant est alors de lancer la fonction `sdl_push_event` sur le premier élément de la liste, dont le délai effectif est le plus court. Nous pouvons alors supprimer l'élément. Si la liste n'est pas vide, alors le prochain timer est lancé et ainsi de suite jusqu'à ce que la liste soit vide. Ainsi, chaque évènement devrait être traité séparément.

3.3 Difficultés rencontrées

Nous avons eu des difficultés à organiser nos fonctions. Nous avons pensé au départ traiter les différents problèmes dans le `timer_set`. Nous avons découpé le cas ou la liste n'était pas vide en 2 sous-cas, l'un lorsque l'évènement actuel avait un délai inférieur aux éléments de la liste, l'autre quand il était supérieur. Nous nous sommes rendus compte que nous obtenions une implémentation qui ne comparait que deux évènements, c'est pourquoi nous avons pensé à un `add_timer`.

De plus, cette implémentation n'était pas claire donc nous avons découpé le code avec d'autres fonctions (`timer_launcher`, `add_timer`).

Nous avons un problème avec les délais. Nous avons essayé de la corriger avec des `printf` des différents délais à chaque nouvel appel de fonction. Nous avons remarqué que les délais deviennent rapidement très élevé. Il peut s'agir d'un problème de conversion entre les différents temps qui sont pour certains en microsecondes et pour d'autres en millisecondes, ou bien d'un problème

de placement dans la liste, mais à ce jour nous n'avons pas réussi à corriger le problème.

Nous n'avons pas eu le temps de nous pencher sur le `timer_cancel`.