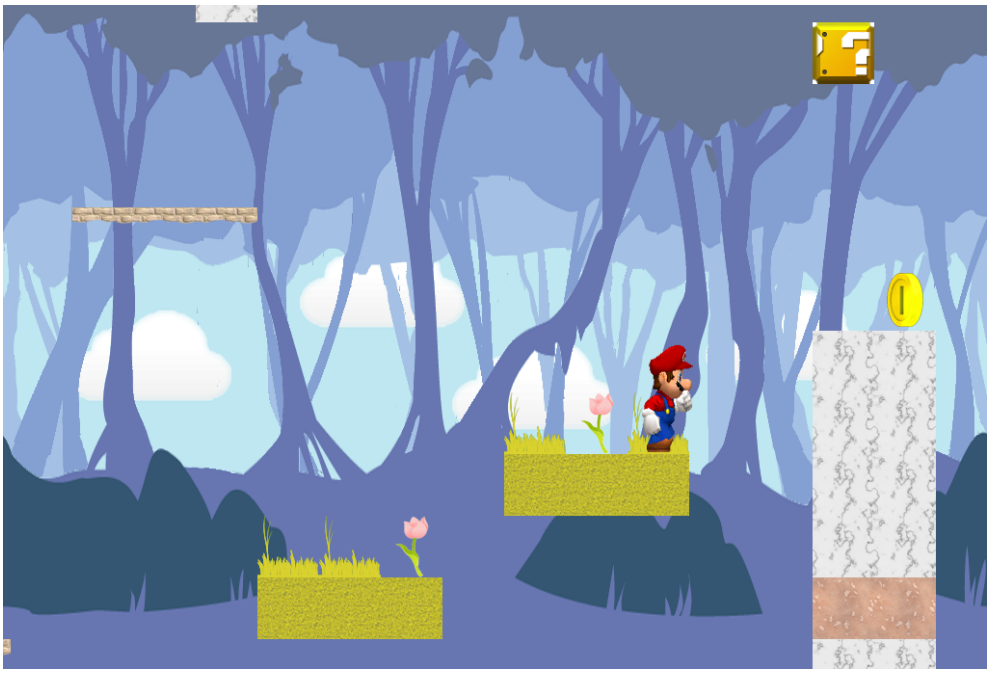
Projet Technologique « Battlefield Mario » : Fiche n°1

1 Objectifs

L'objectif de cette unité d'enseignement est d'apprendre, au travers de la réalisation d'un jeu de plateforme 2D « à la *Mario Maker* », à programmer de manière simple et performante une application complexe.

En particulier, nous étudierons comment exploiter efficacement la bibliothèque graphique SDL, gérer le défilement (*scrolling*) automatique de l'affichage, calculer des collisions au pixel près, maintenir un échéancier d'événements à déclencher, calculer les collisions au pixel près même en présence de nombreux objets dynamiques, etc.



2 Documentation

Le projet s'effectuera en langage C, et s'appuiera sur la bibliothèque `SDL2` pour accélérer l'affichage graphique.

Ce lien vous sera utile pour parcourir les fonctions `SDL2` disponibles par rubrique, ou simplement pour vérifier leurs paramètres :

https://wiki.libsdl.org/APIByCategory

3 Au commencement, il y avait... les oiseaux

Pour s'affranchir, dans un premier temps, de la complexité liée aux nombreux éléments constituant un niveau de jeu de plateforme, nous allons nous concentrer sur le vol d'un oiseau et le défilement du décor derrière lui.

Ouvrez un terminal, et récupérez le répertoire contenant les sources de départ :

```
cp -r ~rnamyst/etudiants/ProjetTechnoMario/TD01 .
```

Puis établissez des liens symboliques vers les répertoires contenant des images et des sons :

```
cd Sources
ln -s ~rnamyst/etudiants/ProjetTechnoMario/images
ln -s ~rnamyst/etudiants/ProjetTechnoMario/sons
```

Normalement, tout est prêt pour la compilation. Tapez `make` et lancez le programme :

```
cd v0
make
./game
```

Si tout se passe bien, une fenêtre s'ouvre et une image composée de nuages blancs sur fond bleu tapisse partiellement la fenêtre.

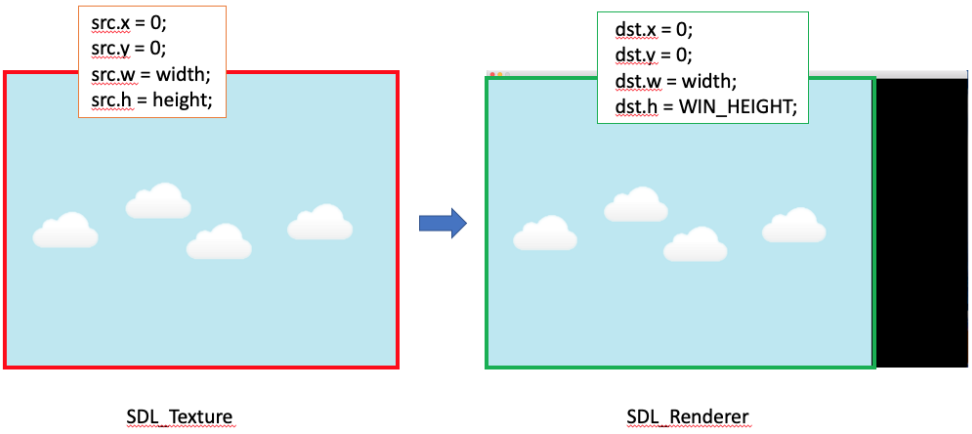
3.1 On veut des nuages !

Nous allons corriger le programme pour tapisser la fenêtre complètement. Commencez par regarder les sources de ce programme.

`main.c` contient la boucle principale du programme. La boucle vérifie si une touche a été appuyée de manière non-bloquante, et demande un rafraichissement de l'écran en appelant la fonction `graphics_render ()`.

Ouvrez le fichier `graphics.c` et analysez le déroulement de l'affichage.

Pour l'instant, la fonction `graphics_render_background` applique une seule fois la `texture` représentant les nuages :



Modifiez la fonction pour appliquer la texture, autant que nécessaire, pour recouvrir complètement le fond. On ne modifiera pas la taille du rectangle de destination : il s'agit simplement d'appliquer une nouvelle fois la texture.

3.2 Entrée en scène de l'oiseau

Écrivez une fonction `graphics_render_object` de profil identique, qui sera chargée d'appliquer la texture de l'oiseau au milieu de l'écran. Notez que la texture représentant un « Toucan » (`lonely_bird.png`) est déjà chargée par le programme. Utilisez la primitive `SDL_RenderCopyEx` afin de faire en sorte que l'oiseau regarde vers la droite.

3.3 Montée et descente

Nous allons ajouter une variable globale `int y_bird` afin de rendre l'altitude de l'oiseau modifiable. Utilisez cette variable dans `graphics_render_object` pour afficher l'oiseau à la distance `y_bird` du haut de la fenêtre.

Dans le programme principal (voir figure 1, page 4), nous allons ajouter le code suivant pour détecter si les touches ↑ ou ↓ sont appuyées, et modifier l'altitude de l'oiseau en conséquence. Veillez à tester que l'oiseau ne sorte pas de l'écran !

Pourquoi n'a-t-on pas testé les touches en utilisant la même méthode que pour la touche *ESC* ?

Lancez le programme avec l'option suivante :

```c
const Uint8 *keystates = SDL_GetKeyboardState (NULL);

for (int quit = 0; !quit;) {
  SDL_Event evt;

  while (SDL_PollEvent (&evt)) {
    ...
  }

  // Check if up/down keys are currently pressed
  int up = keystates[SDL_SCANCODE_UP],
    down = keystates[SDL_SCANCODE_DOWN];

  // FIXME: Move the bird accordingly

  // Refresh screen
  graphics_render ();
}
```

FIGURE 1 – Nouvelle boucle principale avec test séparé des touches UP et DOWN

```
./game -d p
```

Quelle est la phase du programme durant le plus longtemps ? Quelle conclusion en tirez-vous ? Vérifiez en lançant :

```
./game -d p -nvs
```

3.4 Gravité

Un peu comme dans *Flappy Bird*, nous voudrions jouer avec la touche UP pour donner une impulsion à l'oiseau vers le haut, et le voir descendre à cause de la gravité lorsque la touche UP n'est pas activée. Nous allons pour cela ajouter une vitesse verticale `int y_speed` qui sera ajoutée à l'altitude `y_bird` à chaque rafraichissement d'écran.

Lorsque la touche UP est appuyée, retranchez 4 à la vitesse. Tant que la vitesse est inférieure à 10, ajoutez 1 à la vitesse à chaque rafraichissement d'écran. Testez pour ajuster ces paramètres.

3.5 Vitesse horizontale

Pour retrouver le look des jeux d'antan, nous allons ajouter plusieurs plans de décors entre l'oiseau et les nuages. De plus, pour donner une impression de mouvement vers la droite, nous allons faire défiler ces plans dans l'autre sens : vers la gauche.

Ajouter une variable `x_speed` initialisée à la vitesse de votre choix (i.e. nombre de pixels horizontaux parcourus à chaque rafraichissement d'écran). Contrairement à la vitesse verticale, nous n'allons pas modifier la position `x` de l'oiseau avec cette vitesse.

Ajoutez une fonction

```
graphics_render_scrolling_trees (SDL_Texture *tex, int factor)
```

qui sera successivement appelée pour afficher les différents plans arborés. Le paramètre `factor` indiquera la vitesse de défilement du plan : Si `factor` $= 1$, alors le plan défile vers la gauche à la vitesse `x_speed`. Si `factor` $= 2$, alors le plan défile 2 fois moins vite. Etc.

À ce stade, vous devriez obtenir un défilement des trois plans d'arbres donnant une impression de profondeur assez plaisante.

4 Gestion de plusieurs objets mobiles

Si l'on souhaite gérer davantage d'objets mobiles qu'un simple oiseau, il n'est plus raisonnable d'ajouter autant de variables globales, ni même de modifier la routine `graphics_render` pour gérer toutes les animations particulières.

4.1 Objets et sprites

Nous allons définir deux nouveaux types (qui seront des structures C) : les objets et les sprites.
Un sprite contient tout ce qui a trait à la représentation graphique d'un objet :

— la texture associée, ses dimensions
— les dimensions d'origine de la texture
— l'orientation d'origine
— les dimensions d'affichage
— etc.

Un objet représente un élément mobile du jeu qui possède ses propres coordonnées, un pointeur vers sa représentation (un sprite donc, qui peut-être le même pour plusieurs objets), un état, etc.

La figure 2 résume cette organisation.

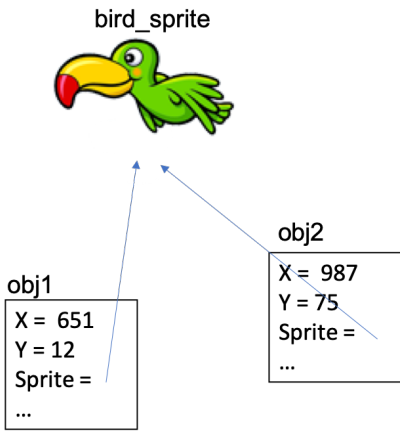


FIGURE 2 – Sprites et Objets

4.2 Sprites

Ajoutez un module d'interface `sprite.h` qui s'occupera du chargement des images et de la fabrication des textures pour chaque sprite du jeu. La figure 3 vous suggère une interface simple pour commencer.

```c
#ifndef SPRITE_IS_DEF
#define SPRITE_IS_DEF

#include <SDL.h>

typedef struct {
  SDL_Texture *texture;
  int native_width, native_height;
  int display_width, display_height;
  int original_direction;
  ...
} sprite_t;

extern sprite_t bird_sprite;

// Initialize sprite_t global variables for each sprite
void sprite_init (void);

// Destroys
void sprite_clean (void);

#endif
```

FIGURE 3 – Interface du module sprite.h

Écrivez le code correpondant dans `sprite.c` en implémentant les fonctions d'initialisation et de destruction des sprites du jeu.

4.3 Objets

De la même manière, nous allons définir une interface pour gérer les objets de manière générique. En fonction du type des objets, il sera pratique de disposer d'une fonction d'« animation » (de type `animate_func_t` [1]) singulière aux objets du même type, qui sera appelée systématiquement à chaque itération. Par exemple, la fonction d'animation pour un objet de type « oiseau » sera de modifier sa vitesse verticale en lui appliquant la gravité, puis de modifier sa coordonnée verticale en fonction de la nouvelle vitesse. La figure 4 donne un cadre permettant de définir des objets mobiles.

L'idée est que lors de l'initialisation de ce module (cf fonction `object_init`), on enregistre dans un tableau indicé par les types d'objets, les routines d'animations pour chaque type d'objets supportés (pour l'instant il n'y a que le type `OBJECT_TYPE_BIRD`, mais à terme il y aura des missiles, des explosions, etc.) Ainsi dans un premier temps, en vous basant sur les seuls types

1. La ligne 31 de la figure 4 permet de définir un nouveau type `animate_func_t` qui est un pointeur de fonction dont la signature est la suivante : `int f(dynamic_object_t *obj)`

fournis par la figure 4 vous pouvez définir un tableau de taille 4, contenant chacun un pointeur vers les fonctions d'animation idoines.

Pourquoi fait-on ça ? Parce que nous aurons bientôt à gérer une liste permettant de parcourir tous les objets mobiles du jeu, et qu'il sera bien pratique de pouvoir parcourir cette liste en exécutant simplement le code suivant pour animer les objets :

```
1 for_all_objects (obj)
2 {
3   animate_func_t func = object_class[obj->type].animate_func;
4
5   if (func != NULL)
6     func (obj);
7 }
```

Écrivez le code correpondant dans `object.c`.

4.4 Putting it all together

Définissez un module `bird.h`, `bird.c` dans lequel vous écrirez la fonction d'animation `animation_bird_onestep (dynamic_object_t *obj)`. Cette fonction sera appelée à chaque rafraîchissement de l'écran.

L'oiseau étant un objet particulier commandé directement par le joueur, nous allons également ajouter une fonction

```
void animation_bird_moves (dynamic_object_t *obj, int up, int down)
```

qui sera appelée depuis le `main`.

Créez un objet `bird` dans `main.c` et modifiez la fonction

```
void graphics_render_object (dynamic_object_t *obj)
```

dans le module `graphics`.

À la fin, tout doit fonctionner comme avant !

5 Objets animés

Examinez le fichier `bird.png` et remarquez qu'elle contient une série d'image décomposant le vol d'un oiseau. Ici, la série contient 8 images, mais pour obtenir un cycle d'animation complet, il faut d'abord afficher les 8 images dans l'ordre de lecture habituel, puis les afficher dans l'ordre inverse.

Il va maintenant s'agir de modifier le module `sprite` afin qu'un objet de type `sprite_t` contienne toutes les informations nécessaires : nombre d'images, nombre d'étapes d'animation, taille d'une image, etc.

Il faudra également modifier le module `object` de manière à ce que chaque objet mobile « se souvienne » de l'étape d'animation à laquelle il est censé s'afficher la prochaine fois...

6 Et voilà

Nous avons maintenant un oiseau qui vole fièrement !

```c
#ifndef OBJECT_IS_DEF
#define OBJECT_IS_DEF

#include <SDL.h>
#include "sprite.h"

enum {
  OBJECT_TYPE_BIRD,
  OBJECT_TYPE_MISSILE,
  OBJECT_TYPE_EXPLOSION,
  OBJECT_TYPE_TEXT,
  __OBJECT_TYPE_NUM
};

enum {
  OBJECT_STATE_NORMAL,
  OBJECT_STATE_IN_AIR,
  OBJECT_STATE_DESTROYED
};

typedef struct _dyn_obj {
  int type;        // can ne OBJECT_TYPE_BIRD, ...
  int state;       // can be OBJECT_STATE_NORMAL, ...
  int x, y;        // position
  int xs, ys;      // speed
  int direction;   // 0 = left, 1 = right
  sprite_t *sprite;
  ...
} dynamic_object_t;

typedef int (*animate_func_t)(dynamic_object_t *obj);//
typedef void (*timer_func_t)(dynamic_object_t *obj);
...

typedef struct {
  animate_func_t animate_func;
} object_type_t;

extern object_type_t object_class [];

// Initialize the object_class array
void object_init (void);

// Initialize fields of object obj
void object_object_init (dynamic_object_t *obj, sprite_t *sp, int type, int x, int y);

#endif
```

FIGURE 4 – Interface du module object.h