Tutoriel : Tile Mapping

Table des matières

Tile Mapping

Ce tutoriel va s'intéresser à la création de jeux de plate-forme, ou jeux en « vue de dessus ». Il présentera et expliquera une technique très utilisée pour ce genre de jeux : le Tile Mapping. Les différents exemples seront proposées en C avec SDL, bien que le but premier soit la compréhension du concept qui permettra de le développer avec n'importe quel langage et n'importe quelle librairie graphique.

Pour comprendre ce tutoriel, vous aurez besoin :

- d'avoir bien assimilé les bases de la programmation en C ;
- d'avoir déjà utilisé la SDL.

Présentation générale

Bienvenue dans la première partie de ce tutoriel.

Dans cette partie, nous allons présenter le Tile Mapping, et faire un petit programme qui affiche un petit monde simple de deux façons différentes :

- avec des nombres directement dans le code ;
- avec un fichier texte comme modèle.

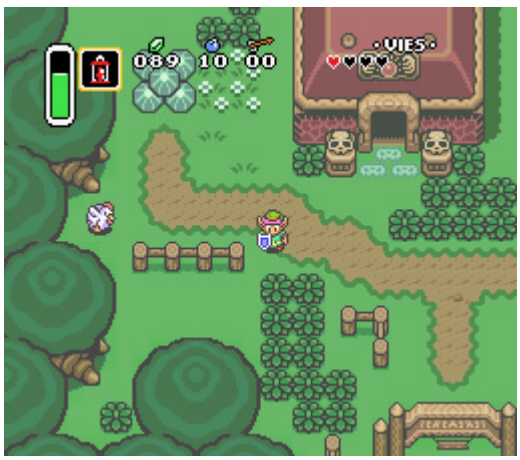
Problématique

Présentation

Vous avez déjà tous joué à ce qu'on appelle des jeux de plateforme en 2D. Il s'agit de jeux où un personnage court dans un monde, parfois à toute vitesse, saute, monte sur des blocs, et infatigablement continue à courir et à sauter... Pendant que vous courez, l'écran défile. Le monde que vous parcourez peut être plus ou moins grand.

Un exemple très connu de jeu de plateforme, duquel nous allons nous inspirer, est Super Mario Bros.

Vous avez surement déjà surement joué aussi a des jeux vus de dessus, comme ce bon vieux Zelda.



Dans ce tutoriel, nous allons essayer de voir comment de tels jeux sont faits. Comment le monde est mis en place et comment faire défiler l'écran.

Comment, avec SDL, peut-on arriver à faire un tel type de jeu ? Comment avoir quelque chose de rapide et d'efficace ?

Cheminement du tutoriel

Ce tutoriel devrait grandir au fur et à mesure des versions. Voici ce qu'il pourrait enseigner au fur et à mesure des versions.

Présentation des techniques.

Les jeux de plateforme ont très rapidement adopté la technique du Tile-Mapping, depuis leur plus jeune âge. Je présenterai tout d'abord rapidement une technique pleine d'inconvénients, puis nous passerons sur la technique du tile mapping.

Création d'un mini monde avec des chiffres

Nous verrons comment créer un petit monde, d'un seul écran, qui ne bouge pas, grâce a un tableau de chiffres.

Mise en place de quelques propriétés, isolement du code

Pour un jeu de plateformes, il sera important de définir où est le sol, où est le ciel, de façon à ce que par la suite, notre personnage puisse évoluer dans le monde logiquement.
Pour un jeu vu de dessus, il sera important de savoir ou on a le droit de marcher, et ou on n'a pas le droit.
Les exemples fournis seront découpés en couches de façon à bien isoler la gestion du monde du reste, et qu'il soit facile, avec une seule fonction, d'afficher un niveau.

Scrolling

Nous verrons ensuite comment faire défiler l'écran (on parle de scrolling), c'est à dire comment faire bouger tout le décor de façon à ce que la caméra suive un personnage, et que le fond défile derrière lui. Tout cela sera également très facile à manipuler au niveau du code.

Insertion d'un personnage

Nous verrons finalement en deuxième partie, comment insérer un personnage dans un décor, comment faire en sorte que la caméra le suive automatiquement et comment faire en sorte qu'un mur l'arrête (collisions).

Evolution

A l'heure ou j'écris ces lignes, le plan futur de ce tuto n'est pas encore écrit, mais nous pourrons envisager les points suivants (en fonction de vos commentaires)

En vrac :

- des « tiles » animées ;
- le scrolling en plusieurs couches (derrière, ça défile moins vite que devant) ;
- des objets qu'on pourrait ramasser ;
- des ennemis qu'on pourrait insérer ;
- plusieurs personnages, avec une caméra intelligente ;
- des blocs cassables ;
- des pentes, des échelles ;
- etc.

Technique de l'image figée

Avant de parler « Tile Mapping », voici une technique qui pourrait être utilisée pour faire un jeu de plateforme.

L'idée qui vient à l'esprit tout de suite est de dessiner son monde sous un logiciel de dessin, « Paint » par exemple. L'idée serait de charger l'image au démarrage du programme, de l'afficher en tant que fond d'écran, puis ensuite, d'afficher un Mario par dessus. Les inconvénients sont les suivants.

C'est coûteux en mémoire

En effet, une grande image, c'est parfois plusieurs mégaoctets de mémoire. Si vous voulez faire un grand monde, multipliez par le nombre d'image nécessaires, et vous obtiendrez une utilisation mémoire inacceptable...

C'est inexploitable

Le plus gros inconvénient est que c'est inexploitable. En effet, si vous affichez votre image de fond, puis que vous affichez Mario, pouvez vous dire facilement si Mario est sur une plateforme ? Dans l'air ? Dans un mur ? Que s'il avance d'un pas, il tombe ?
Et non... Vous pouvez éventuellement vous en sortir sur une image ou le fond est uni, mais si vous avez une image de fond comme CastleVania ci dessous, vous ne pouvez pas vous en sortir de cette façon...

Cette technique a trop d'inconvénients pour être utilisée, je voulais en parler car c'est souvent la première idée qui vient, de "dessiner" son monde, mais nous allons l'oublier, et passer enfin à la technique dont je vous parle depuis tout à l'heure...

Présentation du tile mapping

Avant de définir ce qu'est le Tile Mapping, nous allons ensemble regarder quelques images de jeux connus. Comme on dit qu'un schéma en dit plus qu'un long discours, cela devrait nous aider. :)

Quelle est la particularité des cartes de ces jeux ?
Et bien nous pouvons constater que des motifs se répètent. En effet, les sols sont des briques identiques, collées les unes à coté des autres.

Mieux que ça, on peut remarquer la régularité parfaite de la chose : les points d'interrogation de Mario sont exactement au dessus des briques de sol.

Pareil, dans Zelda, que nous voyons en dessous, il y a des motifs identiques qui se répètent avec une grande régularité.

Le concept de Tile Mapping est de coller côte à côte des "tiles" (c'est à dire des tuiles en anglais) dans une zone régulière. Pour cela, nous subdivisons l'écran en une grille régulière, et nous mettrons un carreau dans chaque case.

Vous voulez voir la grille ?

Si on regarde bien cette dernière image, et qu'on oublie Mario, l'ennemi, l'étoile, et les nuages, qui sont des **sprites**, nous avons affaire à un décor très régulièrement placé. Chaque brique s'emboîte parfaitement dans la grille.

Comme déjà dit, chacune de ces petites briques est appelée tuile, ou tile.

Il y a les tiles uniques, comme les briques et les points d'interrogation, et les tiles composés, comme le pot de fleur, qui est plus gros qu'une case. Cependant, ce pot de fleur, bien qu'il semble être un objet unique, sera vu par la machine comme 8 cases infranchissables...

L'avantage d'une telle méthode est donc qu'au lieu de définir le monde (considérons qu'il ne bouge pas) par une grande image, on le définit par une grille de 13*15 cases.

Coût mémoire

Nous disions que nous définissons l'image d'au dessus par 13*15 cases.
Cela fait 195 cases.

Combien y a-t-il de tiles différents ?
Sur notre image, on a :

- le bloc ciel ;
- le bloc sol ;
- le point d'interrogation ;
- quatre tiles différents pour le pot de fleurs.

... moins d'une dizaine...

Nous pouvons imaginer un tableau de 13 * 15 cases qui contiennent un nombre. Si le nombre est 0, on met du "ciel", si le nombre est 1, on met un bloc cassable, si c'est 2, on met un '?', 3 le bord supérieur gauche du pot, 4 le bord supérieur droit, 5 le bord gauche, 6 le bord droit, 7 le sol d'en bas, etc.

On définit, pour notre Mario, le tableau suivant :

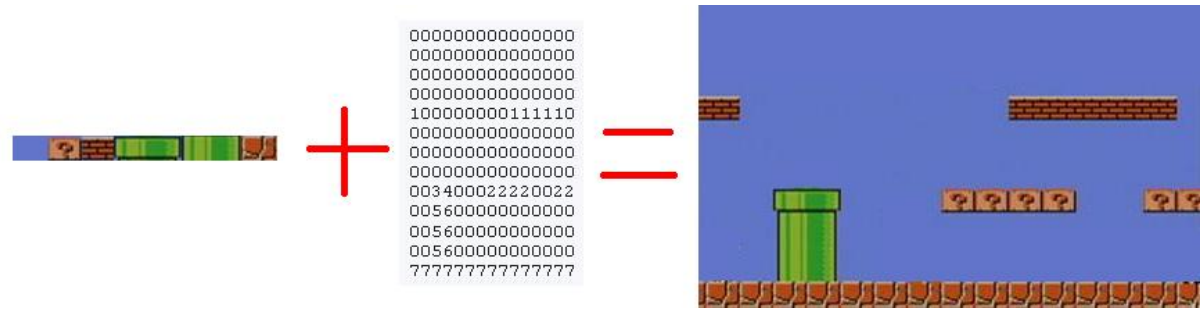
```
000000000000000
000000000000000
000000000000000
000000000000000
100000000111110
000000000000000
000000000000000
000000000000000
003400022220022
005600000000000
005600000000000
005600000000000
777777777777777
```

Ce tableau de nombre décrit parfaitement notre monde, car il nous dit, pour chaque case, quel bloc mettre.
Vous suivez toujours ?

Du coup, avec :

- quelques tiles ;
- un tableau de nombres.

On définit un monde ! Schématiquement, cela donne :



La partie de gauche s'appelle "TileSet". Elle contient les différents carreaux a poser. Ce sont les Tilesets qui définissent le graphisme. Dans mon cas, j'ai fait un tileset d'une seule ligne, mais comme les jeux contiennent quand même davantage de tiles, on définit souvent un tileset sur plusieurs lignes.

Vous trouverez de nombreux exemple sur google image en cherchant "tileset" !
La partie du milieu est le tableau de correspondance.
La partie de droite est bien sur le résultat final.

Revenons maintenant au coût mémoire.

Pour faire mon monde de Mario, j'ai besoin du tileset (une petite image en soi), et du tableau.
Si je considère que je n'aurai pas plus de 256 tiles différents (je tape très large, et c'est souvent le cas), je peux compter 1 octet par case de mon tableau.
Avec mon tableau de 13*15, j'ai moins de 200 octets .... C'est très petit.

Maintenant, supposons un monde entier de Mario (qui défile). Imaginons le monde déplié ainsi :

Combien il y a-t-il de cases la dedans ? A la louche je dirais 300 en largeur, et 20 en hauteur.
300 * 20 = 6000 octets.

Voilà, le monde tout entier tient sur une image tileset petite, + 6 Ko de données : une cacahuète quoi...

Et avec ça, on fait un monde grand.
C'est ainsi qu'ont procédé les consoles 8 et 16 bits, qui n'avaient pas beaucoup de mémoire.

Imaginez que si on avait codé tout le monde sous forme d'une graaaande image, on en aurait eu pour des dizaines de MégaOctets, pour une seule "texture", ce qui aurait bien chargé la carte graphique, et qui, outre ceci, aurait été inexploitable par la suite. (nous verrons les avantages du Tile Mapping lorsque nous parlerons de collisions avec le décor.)

Code exemple

L'algorithme n'est pas complexe. Nous définissons, une fois pour toutes, une longueur et une hauteur de tile (qui restera fixe).

Puis nous faisons un double `for` (i,j) sur le tableau, et nous "blittons" le bon tile à la position (i*largeur, j*hauteur).

Voici l'exemple suivant en C qui reconstruit le petit monde de mario (juste la partie qu'on a étudié)

Téléchargez et dézippez l'ensemble des fichiers de ce tutoriel ci dessous :

Tous les fichiers (http://fvirtman.free.fr/sdz/tilesprogs.zip)

Explication sur les programmes

Vous pouvez constater que le fichier téléchargé contient plusieurs programmes. Je vous dirais au fur et à mesure du tutoriel quel programme ouvrir.

Chaque programme a été fait avec Visual C++ 2008 express, mais pourra être compilé avec d'autres versions, avec code::blokcs, gcc, etc...
Si vous ouvrez le répertoire prog1, en dessous, vous avez un autre répertoire prog1, ainsi qu'un fichier .sln. Si vous avez visual C++, double cliquez sur le sln : le projet d'ouvre et est prêt à compiler.
Sinon, ouvrez le sous répertoire prog1. Vous voyez un .vcproj (également pour Visual C++), et les sources et images utilisées.
Tous les projets sont faits de la même façon.

> Ouvrez maintenant le projet prog1.

Vous pouvez le compiler, et le lancer, ça doit marcher tout seul. (je rappelle que vous devez avoir de bonnes bases en C, et avec la librairie SDL pour poursuivre ce tutoriel).

prog1.c

```
#include <SDL/SDL.h>

#pragma comment (lib,"sdl.lib")      // ignorez ces lignes si vous ne linkez pas les libs de cette f
açon.
#pragma comment (lib,"sdlmain.lib")

#define LARGEUR_TILE 24  // hauteur et largeur des tiles.
#define HAUTEUR_TILE 16

#define NOMBRE_BLOCS_LARGEUR 15  // nombre a afficher en x et y
#define NOMBRE_BLOCS_HAUTEUR 13

char* table[] = {
"000000000000000",
"000000000000000",
"000000000000000",
"000000000000000",
"100000000111110",
"000000000000000",
"000000000000000",
"000000000000000",
"003400022220022",
"005600000000000",
"005600000000000",
"005600000000000",
"777777777777777"};


void Afficher(SDL_Surface* screen,SDL_Surface* tileset,char** table,int nombre_blocs_largeur,int nom
bre_blocs_hauteur)
{
        int i,j;
        SDL_Rect Rect_dest;
        SDL_Rect Rect_source;
        Rect_source.w = LARGEUR_TILE;
        Rect_source.h = HAUTEUR_TILE;
        for(i=0;i<nombre_blocs_largeur;i++)
        {
                for(j=0;j<nombre_blocs_hauteur;j++)
                {
                        Rect_dest.x = i*LARGEUR_TILE;
                        Rect_dest.y = j*HAUTEUR_TILE;
                        Rect_source.x = (table[j][i]-'0')*LARGEUR_TILE;
                        Rect_source.y = 0;
                        SDL_BlitSurface(tileset,&Rect_source,screen,&Rect_dest);
                }
        }
        SDL_Flip(screen);
}

int main(int argc,char** argv)
{
        SDL_Surface* screen,*tileset;
```

```
        SDL_Event event;
        SDL_Init(SDL_INIT_VIDEO);               // prepare SDL
        screen = SDL_SetVideoMode(LARGEUR_TILE*NOMBRE_BLOCS_LARGEUR, HAUTEUR_TILE*NOMBRE_BLOCS_HAUTE
UR, 32,SDL_HWSURFACE|SDL_DOUBLEBUF);
        tileset = SDL_LoadBMP("tileset1.bmp");
        if (!tileset)
        {
                printf("Echec de chargement tileset1.bmp\n");
                SDL_Quit();
                system("pause");
                exit(-1);
        }
        Afficher(screen,tileset,table,NOMBRE_BLOCS_LARGEUR,NOMBRE_BLOCS_HAUTEUR);

        do  // attend qu'on appuie sur une touche.
        {
                SDL_WaitEvent(&event);
        } while (event.type!=SDL_KEYDOWN);

        SDL_FreeSurface(tileset);
        SDL_Quit();
        return 0;
}
```

Le code n'est pas complexe :
Je veux afficher une image de 15*13 tiles (NOMBRE_BLOCS_LARGEUR et NOMBRE_BLOCS_HAUTEUR dans les #define).
Chaque tile fait 24*16 pixels (définis dans LARGEUR_TILE et HAUTEUR_TILE).

Dans le main, j'initialise SDL, la taille de l'image finale, obtenue en faisant l'opération nombre de cases en X * taille d'un tile, et pareil pour y, bien entendu. :)
Je charge le tileset, et je lance la fonction afficher. J'attends qu'on appuie sur une touche pour quitter.

Dans la fonction afficher, je définis 2 SDL_Rect, celui de destination dont vous avez l'habitude, et celui source qui sera passé en 2e paramètre de SDL_BlitSurface pour un blit Partiel.

Je fixe Rect_source.w et .h une fois pour toutes, car les tiles auront toujours la même largeur et la même hauteur.
Puis ensuite, je fais un double for. Je fixe Rect_dest.x et y à la bonne position (qui dépend de i et de j), puis je définis Rect_dest.x, qui lui dépend directement du nombre correspondant. (nous allons détailler cette ligne ci dessous).
Rect_source.y est lui toujours a 0, car dans mon tileset, tous les tiles partent de y=0.

Détaillons la ligne suivante :

```
Rect_source.x = (table[j][i]-'0')*LARGEUR_TILE;
```

Nous avons le tableau table défini, en dur, en haut du code.
Nous voulons récupérer le chiffre à la colonne i, ligne j.
D'ou l'idée de faire table[i][j].

Cependant, vous remarquerez je j'y mis [j][i] et non [i][j]. Cela vient de la définition même du tableau dans le code.
Prenons l'exemple avec des mots plutôt que des chiffres :

```
char* table[] = {
"Bonjour",
"Salut!!",
"Hello!!",
"Saloute",
"Hola!!!"
}
```

Si vous choisissez table[1], vous avez "Salut", puis table[1][2] pour avoir le 'l' de Salut.
Donc la lecture d'un tableau de chaîne se fait en ligne/colonne, alors que nous attendons colonne/ligne dans un repère 2D, d'où la transposée [j][i] au lieu de [i][j].

Maintenant, deuxième soucis, pourquoi est ce que je fais -'0' ?
Si je prends table[j][i], je tombe sur un nombre, mais sur un caractère, sur le code ASCII de '1', celui de '0' etc...

Or le code ASCII de '0' vaut 48. Moi je ne veux pas 48, je veux 0. En soustrayant '0' à un chiffre en ASCII, on obtient le chiffre réel. C'est une astuce classique, sachant que les chiffres sont contiguës dans la table ASCII.

Et voilà, nous avons notre monde de Mario, à partir du tileset et de la table de correspondance. Heureux ?

(J'ai peut être inversé 2 tiles dans mon dessin, mais bon...)

Respirez, et repensez à ce concept à tête reposée. L'essentiel est surtout de comprendre l'idée. Le code va peu à peu évoluer, et surtout se retrouver enfermé dans un autre fichier de façon à vous fournir des fonctions puissantes et simples à utiliser.

Propriétés des tiles

> Dans ce chapitre, nous allons voir comment lire un niveau depuis un fichier texte, et nous allons aussi parler des propriétés des tiles.

Structure d'un fichier texte pour un niveau

L'inconvénient de l'exemple précédent, c'est qu'on entrait directement le niveau dans le code. Si on veut faire des niveaux supplémentaire, ça devient assez contraignant.

Nous allons donc nous appuyer sur un fichier texte que nous allons structurer.

> Ouvrez le répertoire prog2.

Dedans, vous trouvez un fichier appelé level.txt, ouvrez le avec un bloc note.

```
Tilemapping Version 1.0
#tileset
tileset1.bmp
8 1
tile0: vide
tile1: plein
tile2: plein
tile3: plein
tile4: plein
tile5: plein
tile6: plein
tile7: plein
#level
15 13
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 3 4 0 0 0 2 2 2 2 0 0 2 2
0 0 5 6 0 0 0 0 0 0 0 0 0 0 0
0 0 5 6 0 0 0 0 0 0 0 0 0 0 0
0 0 5 6 0 0 0 0 0 0 0 0 0 0 0
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
#fin
```

Expliquons ce fichier.
Tout d'abord, vous pouvez constater que la première ligne est juste une ligne qui donne la version.

Ensuite, nous voyons que le fichier est découpé en 2 blocs :

- #tileset ;
- #level.

Le bloc tileset

Sous le bloc #tileset, ce sont les informations sur le tileset. Avec tout d'abord le nom du tileset à charger : tileset1.bmp.

Ensuite, la ligne suivante contient "8 1". Je définis simplement combien on a de tiles en X, et combien en Y. L'image contient 8 tiles, tous sur la même ligne, donc 8 en X, 1 en Y.
Notez que grâce à la taille de l'image, et du nombre de tiles lus, on peut calculer la largeur et la hauteur d'un tile avec une simple division.

Ensuite, voici la nouveauté de ce chapitre. Comme je l'ai évoqué plus haut, il va falloir définir pour chaque tile s'il est plein ou vide, pour les futurs calculs du jeu.

Typiquement, pour un Mario, seul le premier tile (du ciel) est vide : les personnages pourront s'y promener. Tous les autres sont pleins : on ne pourra pas les traverser.
Pour un Zelda, on aurait en tiles vides tous les sols sur lesquels on peut marcher, et en tiles pleins tous les tiles de rochers, d'arbres, de murs.

C'est ce qu'on appellera les propriétés des tiles. On pourra plus tard en mettre d'autres.

Le bloc level

Le bloc level contient d'abord le nombre de tiles en x et y du level (15 et 13), puis directement le tableau qui définit le level, comme dans le premier chapitre.

Notez cependant que cette fois ci, je mets des espaces entre chaque chiffre. Pourquoi ? Tout simplement parce qu'avec la version du chapitre précédent, nous étions bloqués si nous avions plus de 10 tiles.

Ici, pas de soucis, on pourra très bien avoir 200 tiles et écrire :

```
0   150  1 8
101   6  6 2
```

La balise "fin" explicitera la fin du fichier.

Structure dans le programme

Maintenant que nous avons vu comment était codé notre fichier texte, voyons comment nous allons ranger ça en mémoire.

```
typedef unsigned char tileindex;

typedef struct
{
        SDL_Rect R;
        int plein;
} TileProp;

typedef struct
{
        int LARGEUR_TILE,HAUTEUR_TILE;
        int nbtilesX,nbtilesY;
        SDL_Surface* tileset;
        TileProp* props;
        tileindex** schema;
        int nbtiles_largeur_monde,nbtiles_hauteur_monde;
} Map;
```

La structure Map

Regardons d'abord la structure Map, qui contiendra tout pour nous afficher le monde.

- LARGEUR_TILE, HAUTEUR_TILE : les noms parlent d'eux mêmes.
- nbtilesX,nbtilesY : nombre de tiles dans le tileset, en x et y. Dans notre cas, ce sera 8 et 1.
- nbtiles_largeur_monde,nbtiles_hauteur_monde : garderont les 15 et 13 de notre exemple.
- tileset : lien vers la surface tileset chargée.
- props : Un tableau de propriétés pour chaque tile. Sa taille sera nbtilesX*nbtilesY. L'élément du tableau est une autre structure, TileProp, que nous allons voir plus bas.
- schema: Tableau à deux dimensions, qui contiendra tous les chiffres de la partie "level". Sa taille sera nbtiles_largeur_monde en X, nbtiles_hauteur_monde en Y.

Le type tileindex

schema est un tableau vers des types "tileindex" que j'ai défini plus haut.
Dans notre cas, j'ai défini tileindex à "unsigned char", car je n'aurai pas plus de 256 tiles.
Si vous en avez davantage, changez juste le typedef en "unsigned short".
Si les "unsigned short" ne suffisent pas, on pourrait mettre "unsigned long", mais en réalité, plus de 65536 tiles dans la même map, c'est une folie...

La structure TileProp

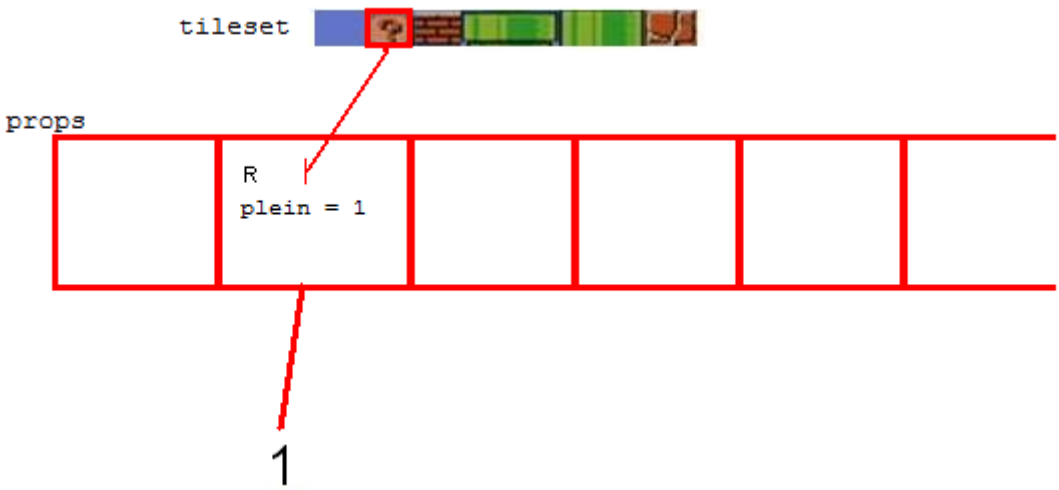
La structure TileProp contient pour l'instant deux choses :

- un rectangle R ;
- la propriété plein/vide.

Pourquoi stockons nous un SDL_Rect ?
Dans l'exemple du fichier précédent, vous avez pu constater dans la fonction Affichage que pour tous les tiles à afficher, je calculais un rectangle source sur le tileset, à chaque fois.
Comme on a beaucoup de tiles identiques, on recalculait plein de fois les mêmes.

Je propose de les calculer une fois pour toutes, et de les stocker. Ainsi, pour les futurs blits, on a déjà le rectangle source, il ne reste plus qu'à poser le tile à la bonne destination.

Schématiquement, on peut voir le lien ici entre le tableau de propriétés, et le tileset. Je n'ai rempli que la case 1, mais on aura la même chose avec toutes les autres cases.



Code exemple

Voyons maintenant une illustration de tout cela.
Ouvrez le projet "prog2".

Compilez le, lancez le. Vous devriez avoir le même résultat que le programme précédent... mais avec un code plus complexe !

> *Pourquoi faire compliqué quand on peut faire simple ?*

Parce que le code précédent était inexploitable.
On peut envisager la métaphore suivante :
Si vous voulez construire une maison, la méthode la plus rapide est de poser tous les murs brutalement, puis le toit.
Par contre, si on vous dit "maintenant, ou on fait passer les tuyaux d'eau ? Les cables ?", et bien vous n'avez plus qu'à démolir.
Alors que si vous avez construit votre maison en prévoyant les goulottes, vous pourrez rajouter l'électricité par la suite, même si avant ça, les maisons avec et sans avoir prévu les goulottes se ressemblent.

Etudions le programme.

Regardons d'abord le fmap.h

```
#include <sdl/sdl.h>

#pragma comment (lib,"sdl.lib")      // ignorez ces lignes si vous ne linkez pas les libs de cette f
açon.
#pragma comment (lib,"sdlmain.lib")

typedef unsigned char tileindex;

typedef struct
{
        SDL_Rect R;
        int plein;
        // tout ce que vous voulez...
} TileProp;

typedef struct
{
        int LARGEUR_TILE,HAUTEUR_TILE;
        int nbtilesX,nbtilesY;
        SDL_Surface* tileset;
        TileProp* props;
        tileindex** schema;
        int nbtiles_largeur_monde,nbtiles_hauteur_monde;
} Map;

Map* ChargerMap(const char* fic);
int AfficherMap(Map* m,SDL_Surface* screen);
int LibererMap(Map* m);
```

On retrouve les structures étudiées plus haut, ainsi que 3 fonctions seulement :

- ChargerMap ;
- AfficherMap ;
- LibererMap.

Si vous n'êtes pas curieux, ce sont des fonctions magiques.
A la première, vous passez le fichier level.txt, et elle vous remplit une Map qu'elle vous donne.
A la seconde, vous donnez la map, et l'écran "screen" ou vous voulez l'afficher, et elle l'affiche.
A la troisième, vous donnez la map, elle nettoie proprement.

Maintenant, regardons le main, dans prog2.c

```
#include "fmap.h"


int main(int argc,char** argv)
{
        SDL_Surface* screen;
        SDL_Event event;
        Map* m;
        SDL_Init(SDL_INIT_VIDEO);               // prepare SDL
        screen = SDL_SetVideoMode(360, 208, 32,SDL_HWSURFACE|SDL_DOUBLEBUF);
        m = ChargerMap("level.txt");
        AfficherMap(m,screen);
        SDL_Flip(screen);
        do
        {
                SDL_WaitEvent(&event);
        } while (event.type!=SDL_KEYDOWN);
        LibererMap(m);
        SDL_Quit();
        return 0;
}
```

Regardez la simplicité d'utilisation : je charge la map, je l'affiche, j'attends qu'on appuie sur une touche, et je la libère avant de la quitter.

Si vous n'êtes pas curieux donc, voila comment faire simple.

Pour les curieux, regardons fmap.c

```
#define _CRT_SECURE_NO_DEPRECATE   // pour visual C++ qui met des warning pour fopen et fscanf : auc
un effet negatif pour les autres compilos.
#include <string.h>
#include "fmap.h"

#define CACHE_SIZE 5000

SDL_Surface* LoadImage32(const char* fichier_image)
{
        SDL_Surface* image_result;
        SDL_Surface* image_ram = SDL_LoadBMP(fichier_image);    // charge l'image dans image_ram en
RAM
        if (image_ram==NULL)
        {
                printf("Image %s introuvable !! \n",fichier_image);
                SDL_Quit();
                system("pause");
                exit(-1);
        }
        image_result = SDL_DisplayFormat(image_ram);
        SDL_FreeSurface(image_ram);
        return image_result;
}

void ChargerMap_tileset(FILE* F,Map* m)
{
        int numtile,i,j;
        char buf[CACHE_SIZE];  // un buffer, petite mémoire cache
        char buf2[CACHE_SIZE];  // un buffer, petite mémoire cache
        fscanf(F,"%s",buf); // nom du fichier
        m->tileset = LoadImage32(buf);
        fscanf(F,"%d %d",&m->nbtilesX,&m->nbtilesY);
        m->LARGEUR_TILE = m->tileset->w/m->nbtilesX;
        m->HAUTEUR_TILE = m->tileset->h/m->nbtilesY;
        m->props = malloc(m->nbtilesX*m->nbtilesY*sizeof(TileProp));
        for(j=0,numtile=0;j<m->nbtilesY;j++)
        {
                for(i=0;i<m->nbtilesX;i++,numtile++)
                {
                        m->props[numtile].R.w = m->LARGEUR_TILE;
                        m->props[numtile].R.h = m->HAUTEUR_TILE;
                        m->props[numtile].R.x = i*m->LARGEUR_TILE;
                        m->props[numtile].R.y = j*m->HAUTEUR_TILE;
                        fscanf(F,"%s %s",buf,buf2);
                        m->props[numtile].plein = 0;
                        if (strcmp(buf2,"plein")==0)
                                m->props[numtile].plein = 1;
                }
        }
}

void ErrorQuit(const char* error)
{
```

```
        puts(error);
        SDL_Quit();
        system("pause");
        exit(-1);
}

void ChargerMap_level(FILE* F,Map* m)
{
        int i,j;
        fscanf(F,"%d %d",&m->nbtiles_largeur_monde,&m->nbtiles_hauteur_monde);
        m->schema = malloc(m->nbtiles_largeur_monde*sizeof(tileindex*));
        for(i=0;i<m->nbtiles_largeur_monde;i++)
                m->schema[i] = malloc(m->nbtiles_hauteur_monde*sizeof(tileindex));
        for(j=0;j<m->nbtiles_hauteur_monde;j++)
        {
                for(i=0;i<m->nbtiles_largeur_monde;i++)
                {
                        int tmp;
                        fscanf(F,"%d",&tmp);
                        if (tmp>=m->nbtilesX*m->nbtilesY)
                                ErrorQuit("level tile hors limite\n");
                        m->schema[i][j] = tmp;
                }
        }
}

Map* ChargerMap(const char* level)
{
        FILE* F;
        Map* m;
        char buf[CACHE_SIZE];
        F = fopen(level,"r");
        if (!F)
                ErrorQuit("fichier level introuvale\n");
        fgets(buf,CACHE_SIZE,F);
        if (strstr(buf,"Tilemapping Version 1.0")==NULL)
                ErrorQuit("Mauvaise version du fichier level. Ce programme attend la version 1.0
\n");
        m = malloc(sizeof(Map));
        do
        {
                fgets(buf,CACHE_SIZE,F);
                if (strstr(buf,"#tileset"))
                        ChargerMap_tileset(F,m);
                if (strstr(buf,"#level"))
                        ChargerMap_level(F,m);
        } while (strstr(buf,"#fin")==NULL);
        fclose(F);
        return m;
}

int AfficherMap(Map* m,SDL_Surface* screen)
{
        int i,j;
```

```
        SDL_Rect Rect_dest;
        int numero_tile;
        for(i=0;i<m->nbtiles_largeur_monde;i++)
        {
                for(j=0;j<m->nbtiles_hauteur_monde;j++)
                {
                        Rect_dest.x = i*m->LARGEUR_TILE;
                        Rect_dest.y = j*m->HAUTEUR_TILE;
                        numero_tile = m->schema[i][j];
                        SDL_BlitSurface(m->tileset,&(m->props[numero_tile].R),screen,&Rect_dest);
                }
        }
        return 0;
}

int LibererMap(Map* m)
{
        int i;
        SDL_FreeSurface(m->tileset);
        for(i=0;i<m->nbtiles_hauteur_monde;i++)
                free(m->schema[i]);
        free(m->schema);
        free(m->props);
        free(m);
        return 0;
}
```

Ce fichier n'est pas très complexe si vous savez lire un fichier texte.
Car la fonction ChargerMap ne fait finalement que des fscanf et des fgets.

Notons la fonction LoadImage32 qui va charger l'image, et la mettre au format de votre écran grâce à SDL_DisplayFormat, pour une vitesse de blit optimale.

La fonction ChargerMap_tileset va remplir le tableau, et le rectangle R pour chaque type de tile.

La fonction AfficherMap en est donc simplifiée, puisqu'elle va directement lire le rectangle source, et non le calculer à chaque fois.

A la fin de ce chapitre, vous voyez qu'on peut enfermer la difficulté dans un fichier (fmap.c) et le niveau dans un fichier texte (level.txt).
Finalement, le pilotage de l'ensemble, dans le main, reste très simple.

Amusez vous à modifier le fichier level.txt (par exemple en changeant les chiffres du tableau de schéma pour voir les modifications appliquées en relançant le programme, même sans le recompiler.

Scrolling

Jusqu'à maintenant, on affichait une pauvre image fixe. Nous allons voir ici comment avoir un scrolling, c'est-à-dire un défilement d'écran.
Quand dans mario, vous courez, vous voyez le paysage qui défile derrière vous. C'est ce qu'on appelle le scrolling.

L'idée de l'image géante

Dans ce paragraphe, nous allons parler de l'idée de l'image géante. C'est une idée à laquelle on pense quand on veut faire du scrolling...

Imaginons un monde de mario qui fait 300 tiles de large (le monde entier, du départ jusqu'au drapeau). Chaque tile fait 24 pixels de large.

Si nous voulons dessiner tout le monde d'un coup, nous avons donc besoin d'une image de 300*24 = 7200 pixels de large. Une image géante donc !

L'idée est de blitter l'image géante où il faut, pour que seule la partie "intéressante" apparaisse. Et de la blitter légèrement plus loin à la frame d'après pour qu'on ait l'impression qu'on a bougé.

Mais cette image géante, en mémoire, prendra plusieurs dizaines de Mo. Et encore, si le monde est grand comme une carte de Zelda, ce sera en centaines de Mo que ça se comptera...

Sur un PC puissant, cette technique pourra marcher, même si elle risque de saturer la mémoire graphique (VRAM) mais c'est épouvantablement lourd.

> *Alors pourquoi les consoles comme la NES, très peu puissantes, arrivaient à gérer des scrollings alors qu'elles n'avaient que quelques Ko de mémoire ?*

Tout simplement parce que stocker une image géante n'est pas une bonne idée...

Le fenêtrage

Bien que trop lourde en mémoire, nous n'allons pas oublier notre image géante.
Nous allons pour l'instant juste imaginer qu'elle existe, mais ne pas la stocker en mémoire.

Voici une belle image :

(http://sdz.tdct.org/sdz/medias/uploads.siteduzero.com_files_190001_191000_190176.png)

Elle venait d'un exemple précédent que j'ai mis à jour. Dans l'exemple de la fin de ce chapitre, nous en aurons une autre plus jolie.

> *Qu'est ce que le rectangle rouge en bas à gauche ?*

C'est un rectangle que j'ai rajouté pour l'exemple. Ce rectangle, je vais l'appeler fenêtre.

Cette fenêtre, c'est ce que vous verrez sur votre écran. Cette fenêtre se décalera et vous verrez donc autre chose. Si cette fenêtre glisse vers la droite, alors on aura l'impression d'avancer dans le monde.

Vous voyez le concept ? Seule la partie affichée dans la fenêtre sera affichée sur votre écran.

J'appellerai ce rectangle la **fenêtre du scrolling** et il suffira de déplacer cette fenêtre pour faire défiler le niveau.

Je vous propose un petit travail manuel pour bien vous en rendre compte. Prenez une feuille A4, et découpez, en plein milieu, un rectangle de la taille du rectangle rouge. Posez la feuille sur votre écran sur l'image géante ci dessus. Puis déplacez la. Vous voyez le monde défiler dans le trou que vous avez fait.

Deux repères

Notre monde entier est l'image géante, que nous n'afficherons jamais entièrement, mais qui existe.
Dans ce monde, les coordonnées varient de 0 à .... beaucoup. 10 000 peut être, bien plus encore, si notre monde est grand.
Nous appellerons ça le repère absolu, ou repère global.

Par contre, la partie que nous voyons à l'écran, elle, a toujours la même largeur et hauteur (notre écran) avec ses coordonnés qui vont de 0 à 800 par exemple, jamais plus.
Nous appellerons ça le repère local.

Pour passer de l'un à l'autre, c'est très simple : nous allons définir le point S, de coordonnées xscroll/yscroll pour la fenêtre. C'est le point du coin supérieur gauche de la fenêtre dans le repère global.

Si on a un point dans le repère local, et qu'on veut sa coordonnée dans le repère global, on fait une addition.
Pour passer de global à local, on fait une soustraction.
P_{global} = P_{local} + SP_{local} = P_{global} - S

> Le simple fait de modifier le point S permettra le scrolling.

Voici comment nous allons modifier notre structure Map :

```
typedef struct
{
        SDL_Rect R;
        char plein;
} TileProp;

typedef struct
{
        int LARGEUR_TILE,HAUTEUR_TILE;
        int nbtiles;
        TileProp* props;
        SDL_Surface* tileset;
        tileindex** schema;
        int nbtiles_largeur_monde,nbtiles_hauteur_monde;
        int xscroll,yscroll;
        int largeur_fenetre,hauteur_fenetre;
} Map;
```

Vous pouvez constater, par rapport aux structures de l'exemple d'avant, que seuls 4 paramètres ont été ajoutés : le reste n'a pas bougé.

```
int xscroll,yscroll;
int largeur_fenetre,hauteur_fenetre;
```

Alors ces paramètre sont très simples :
largeur_fenetreet, hauteur_fenetre sont la largeur et hauteur de ma fenêtre de scrolling (la largeur et hauteur du rectangle rouge), et xscroll et yscroll sont la position de son point supérieur gauche, le point S.
Exactement comme un SDL_rect !

> *Mais pourquoi ne pas utiliser un SDL_Rect ?*

Parce qu'un SDL_Rect utilise un x,y en tant que *signed short*, c'est-à-dire qu'il est limité à 32767 pixels.

Or, si notre monde est très très grand, l'image géante imaginée sera possiblement plus grande que cela. Nous utiliserons donc un int qui pourra nous permettre d'aller beaucoup plus loin.

largeur_fenetre et hauteur_fenetre resteront invariants : tout au long du jeu, la taille de la fenêtre d'affichage (ce que vous voyez) restera constante.
Par contre, xscroll et yscroll, eux, changeront.
Et quand il changeront la fenêtre rouge sur l'image géante se déplacera. Concrètement, il y aura scrolling...

> L'idée est donc maintenant de modifier la fonction d'affichage pour qu'elle affiche la partie du monde correspondant a la fenêtre rouge.

première version

```
int AfficherMap(Map* m,SDL_Surface* screen)
{
        int i,j;
        SDL_Rect Rect_dest;
        int numero_tile;
        for(i=0;i<m->nbtiles_largeur_monde;i++)
        {
                for(j=0;j<m->nbtiles_hauteur_monde;j++)
                {
                        Rect_dest.x = i*m->LARGEUR_TILE - m->xscroll;
                        Rect_dest.y = j*m->HAUTEUR_TILE - m->yscroll;
                        numero_tile = m->schema[i][j];
                        SDL_BlitSurface(m->tileset,&(m->props[numero_tile].R),screen,&Rect_dest);
                }
        }
        return 0;
}
```

Vous pouvez constater que la seule différence avec la fonction AfficherMap d'avant, c'est que Rdest.x et Rdest.y sont otés de m->xscroll et m->yscroll

Concrètement, avec cette fonction, je vais afficher TOUTE la grande map (car mes for vont de 0 à nbtiles_largeur_monde et de 0 à nbtiles_hauteur_monde, donc couvrent tout), mais en "décalant" de xscroll et yscroll.
Concrètement, si mon rectangle rouge est à l'endroit ci dessus, je vais quand même tout afficher (90% sera hors de l'écran mais tant pis) y compris le "FRED" qu'on voit en haut : il sera affiché hors écran (donc ignoré) mais on le calculera quand même...

Cette simple soustraction permet le scrolling. En effet, si xscroll évolue positivement, alors, pour chaque tile, Rect_dest.x évoluera négativement : ce qui est normal, car quand le rectangle rouge avance, on a l'impression que les tiles reculent ! Et oui, c'est magique !
Si vous regardez Mario, quand vous courrez dans un monde, les tiles, eux, vont en arrière...

L'exemple qui finira cette partie vous illustrera cela.

Deuxième version

L'inconvénient de la première version est qu'elle va essayer de blitter tous les tiles du niveau. Quand ils seront dehors, ils ne seront pas affichés, mais la machine essayera des les blitter quand même.
Du coup, plus le monde sera grand, plus les for seront longs, et plus la machine tentera de blitter, et plus ça va ralentir...
C'est dommage.

Je propose donc une optimisation.

Voici l'idée : seuls les tiles présents dans le cadre rouge devront être affichés. Les autres seront dehors : inutile de les afficher.
Nous allons donc, au lieu de faire varier notre for entre 0 et m->nbtiles_largeur_monde, le faire varier entre un xmin et un xmax. Pareil pour y.
Ainsi, nous restreignons notre boucle à la seule zone d'affichage.

Il faut donc calculer ces xmin, xmax, ymin et ymax

Quel est le xmin ? C'est la coordonnée de gauche de la fenêtre que divise la taille d'un tile tout simplement.
Et quel est le max ? La coordonnée de droite de la fenêtre que divise la taille d'un tile ...

Cela nous donne immédiatement notre deuxième version de la fonction AfficherMap :

```
int AfficherMap(Map* m,SDL_Surface* screen)
{
        int i,j;
        SDL_Rect Rect_dest;
        int numero_tile;
        int minx,maxx,miny,maxy;
        minx = m->xscroll / m->LARGEUR_TILE-1;
        miny = m->yscroll / m->HAUTEUR_TILE-1;
        maxx = (m->xscroll + m->largeur_fenetre)/m->LARGEUR_TILE;
        maxy = (m->yscroll + m->hauteur_fenetre)/m->HAUTEUR_TILE;
        for(i=minx;i<=maxx;i++)
        {
                for(j=miny;j<=maxy;j++)
                {
                        Rect_dest.x = i*m->LARGEUR_TILE - m->xscroll;
                        Rect_dest.y = j*m->HAUTEUR_TILE - m->yscroll;
                        numero_tile = m->schema[i][j];
                        SDL_BlitSurface(m->tileset,&(m->props[numero_tile].R),screen,&Rect_dest);
                }
        }
        return 0;
}
```

je calcule mon xmin, xmax, ymin, ymax, puis je ne fais varier mes for que dans ces zones-là.
Pour xmin et ymin, je mets -1 car si le fenêtrage est entre deux tiles, il faut que le tile d'avant soit affiché, pour voir le morceau de droite (ou du bas) arriver par la gauche (ou par le haut).

> La taille de la map finale ne ralentira plus rien : en effet, la map peut être grand ou petite, il n'y aura pas plus de calculs.

Troisième version

Avec la deuxième version, la fonction plantera si la fenêtre de scrolling sort de l'espace de l'image géante, car les i,j déborderont du tableau m->schema.

Donc soit on fait attention à limiter le scrolling,
soit on protège la fonction avec un if, soit les deux.

Dans le cas de cette version, si on sort du schéma, on dit que on a des tiles de type 0, à l'infini...

```
int AfficherMap(Map* m,SDL_Surface* screen)
{
        int i,j;
        SDL_Rect Rect_dest;
        int numero_tile;
        int minx,maxx,miny,maxy;
        minx = m->xscroll / m->LARGEUR_TILE-1;
        miny = m->yscroll / m->HAUTEUR_TILE-1;
        maxx = (m->xscroll + m->largeur_fenetre)/m->LARGEUR_TILE;
        maxy = (m->yscroll + m->hauteur_fenetre)/m->HAUTEUR_TILE;
        for(i=minx;i<=maxx;i++)
        {
                for(j=miny;j<=maxy;j++)
                {
                        Rect_dest.x = i*m->LARGEUR_TILE - m->xscroll;
                        Rect_dest.y = j*m->HAUTEUR_TILE - m->yscroll;
                        if (i<0 || i>=m->nbtiles_largeur_monde || j<0 || j>=m->nbtiles_hauteur_mond
e)
                                numero_tile = 0;
                        else
                                numero_tile = m->schema[i][j];
                        SDL_BlitSurface(m->tileset,&(m->props[numero_tile].R),screen,&Rect_dest);
                }
        }
        return 0;
}
```

Code exemple

Voici maintenant le code.
Ouvrez le projet "prog3", compilez le et lancez le.

Appuyez sur les flèches pour faire défiler le paysage !

Expliquons un peu le code.

fevent.h

```
#include <sdl/sdl.h>

typedef struct
{
        char key[SDLK_LAST];
        int mousex,mousey;
        int mousexrel,mouseyrel;
        char mousebuttons[8];
        char quit;
} Input;

void UpdateEvents(Input* in);
void InitEvents(Input* in);
```

C'est ma façon de gérer les events, je mets à jour ma structure Input. J'explique tout cela dans un autre tutoriel (http://www.siteduzero.com/tutoriel-3-145225-simplifier-les-events-avec-sdl.html). fevent.c va avec.

Concentrons nous maintenant sur le main, dans prog3.c

```
#include "fmap.h"
#include "fevent.h"

#define LARGEUR_FENETRE 500
#define HAUTEUR_FENETRE 500
#define MOVESPEED 1

void MoveMap(Map* m,Input* in)
{
	if (in->key[SDLK_LEFT])
		m->xscroll-=MOVESPEED;
	if (in->key[SDLK_RIGHT])
		m->xscroll+=MOVESPEED;
	if (in->key[SDLK_UP])
		m->yscroll-=MOVESPEED;
	if (in->key[SDLK_DOWN])
		m->yscroll+=MOVESPEED;
// limitation
	if (m->xscroll<0)
		m->xscroll=0;
	if (m->yscroll<0)
		m->yscroll=0;
	if (m->xscroll>m->nbtiles_largeur_monde*m->LARGEUR_TILE-m->largeur_fenetre-1)
		m->xscroll=m->nbtiles_largeur_monde*m->LARGEUR_TILE-m->largeur_fenetre-1;
	if (m->yscroll>m->nbtiles_hauteur_monde*m->HAUTEUR_TILE-m->hauteur_fenetre-1)
		m->yscroll=m->nbtiles_hauteur_monde*m->HAUTEUR_TILE-m->hauteur_fenetre-1;
}

int main(int argc,char** argv)
{
	SDL_Surface* screen;
	Map* m;
	Input I;
	InitEvents(&I);
	SDL_Init(SDL_INIT_VIDEO);		// prepare SDL
	screen = SDL_SetVideoMode(LARGEUR_FENETRE, HAUTEUR_FENETRE, 32,SDL_HWSURFACE|SDL_DOUBLEBUF);
	m = ChargerMap("level2.txt",LARGEUR_FENETRE,HAUTEUR_FENETRE);
	while(!I.key[SDLK_ESCAPE] && !I.quit)
	{
		UpdateEvents(&I);
		MoveMap(m,&I);
		AfficherMap(m,screen);
		SDL_Flip(screen);
		SDL_Delay(1);
	}
	LibererMap(m);
	SDL_Quit();
	return 0;
}
```

Dans le main, j'initialise ChargerMap en précisant en paramètres supplémentaires la taille de la fenêtre que je désire. Si vous voulez une fenêtre plus grande, changez simplement les #define en haut de ce fichier.

Dans le while du main, j'appelle une fonction MoveMap qui est au dessus. C'est elle qui va me permettre de commander mon scrolling.
Puis j'affiche la map, je flip et je fais un Delay pour réguler la vitesse.

La fonction MoveMap est très simple :
Je regarde les touches de direction, et en fonction d'elles, je mets simplement à jour les variables xscroll et yscroll. La fonction AfficherMap en tiendra compte.

Notez la partie "limitation", qui empêche la fenêtre de sortir du repère global. Si vous l'enlevez, alors vous pourrez sortir sans soucis.
Et dans la mesure ou la fonction AfficherMap considère que tout ce qui est dehors est le tile 0, alors si vous sortez, vous verrez des "tile 0" à perte de vue. Essayez donc !

Ici, mon tile 0 est un bloc qui se voit bien. Mais on pourrait mettre du ciel, ou donner un autre numéro de tile par défaut...

En ce qui concerne le fichier fmap.c, tout ce qui diffère avec la version précédente, c'est la fonction AfficherMap expliquée juste au dessus.
Il y a juste, à la fin de la fonction ChargerMap, le stockage des variables passées, et une initialisation de xscroll et yscroll à 0.

Ce qu'il faut bien retenir dans le scrolling, c'est que nous "imaginons" une grande image, faite du monde entier, qui peut être très grand ; et nous n'affichons que la partie désirée.
Nous ne stockons pas l'image géante complète en mémoire, mais nous stockons uniquement de quoi en calculer rapidement une partie (celle qui sera visible).

Pour faire défiler l'écran, il **suffit** de changer la valeur des variables xscroll et yscroll. L'affichage affiche ce qu'il faut en conséquence.

Ainsi, un scrolling horizontal et vertical revient uniquement à mettre à jour 2 variables...

Un Editeur

> Si vous regardez le fichier level2.txt du projet prog3 avec un bloc note, vous pourrez constater qu'il y a beaucoup de nombre écrits, et que quelque part, ce n'est pas humain d'écrire tout ça à la main. Et vous avez raison !
> On dit qu'il faut toujours se créer ses propres outils. Voici un éditeur qui permet de créer des fichiers de niveaux.

Utilisation de l'éditeur

L'éditeur se trouve dans le répertoire "Edit1".

Vous trouverez les sources que vous pourrez compiler pour l'utiliser.

Lancement de l'éditeur

L'éditeur ne démarre pas sans paramètres. Vous devez lui passer :

- soit un fichier .bmp (un tileset) pour faire une nouvelle carte ;
- soit un fichier .txt (un level déjà fait) pour en recharger une et la modifier.

Vous pouvez lancer l'éditeur en ligne de commande avec un tel paramètre, ou bien faire glisser un bmp ou un txt sur l'EXE sous Windows : ça revient au même.

Dans le répertoire edit1, il y a des fichiers bmp ou des fichiers txt en exemple pour essayer.

Nouvelle carte.

Si vous lancez une nouvelle carte, l'éditeur va vous demander combien il y a de tiles en x et en y sur votre tileset (à vous de compter). De la il calculera automatiquement leur largeur et hauteur.
Puis vous vous retrouverez dans l'éditeur proprement dit.

Ancienne carte.

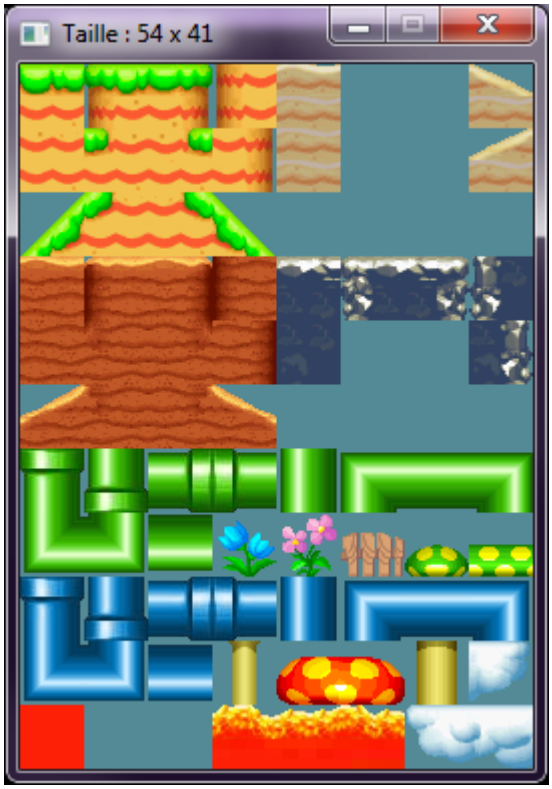
Il chargera simplement une ancienne carte.
Puis vous vous retrouverez dans l'éditeur proprement dit.

La zone d'édition

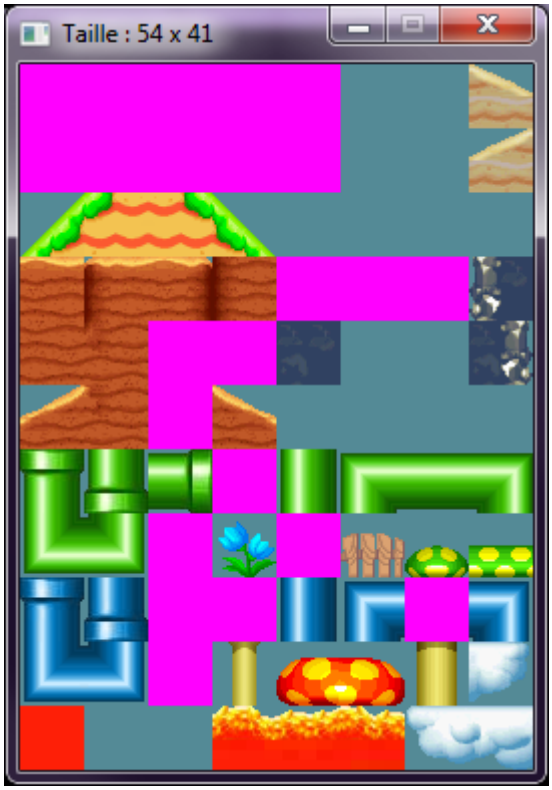
L'éditeur se compose en 2 parties. La partie "choix du tile" qui affiche le tileset, et la partie "level" qui affiche le niveau en cours.
Au départ, c'est la partie "level qui s'affiche"

Pour passert d'une partie à une autre, cliquez sur le bouton droit de la souris à tout moment.

Partie "choix du tile"

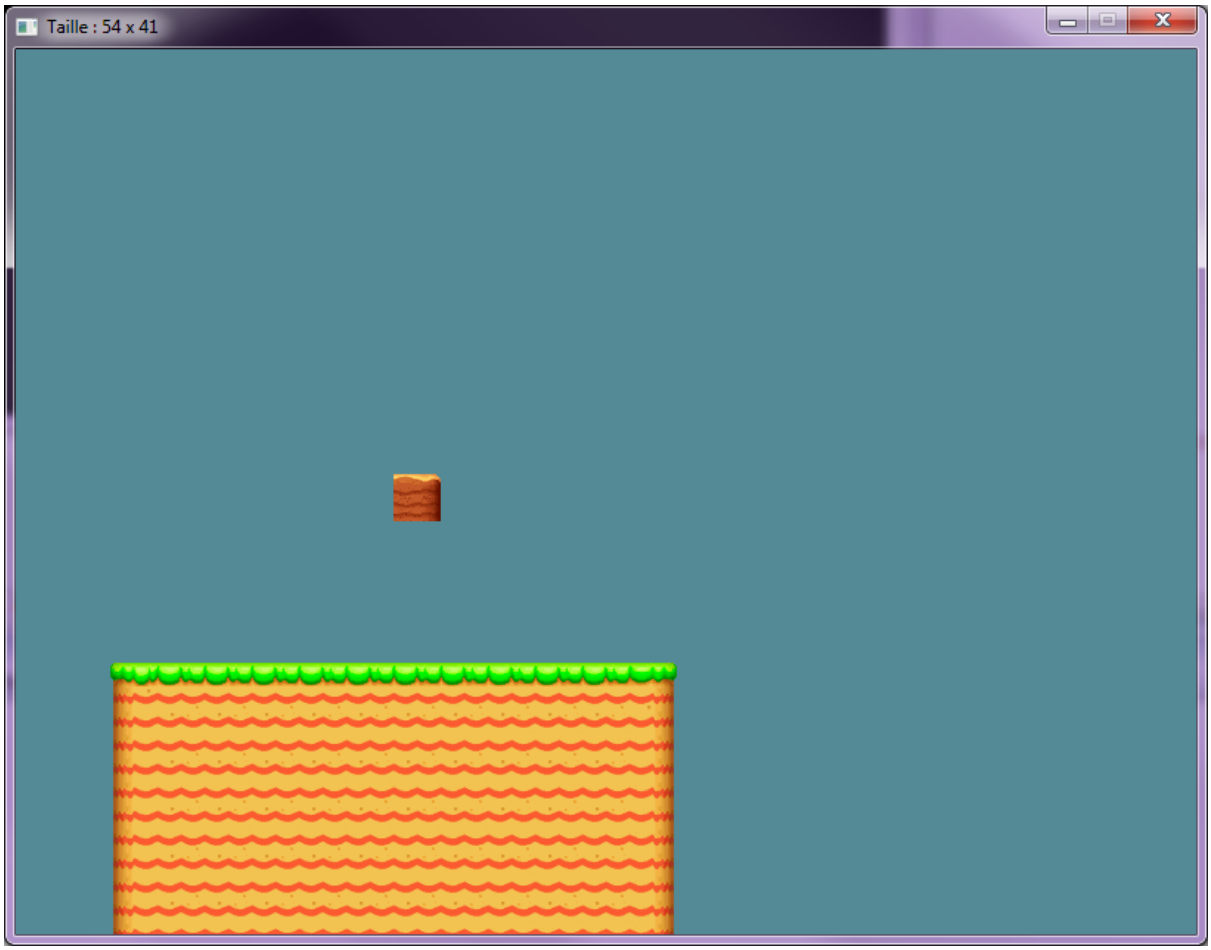


Dans cette partie, vous avez le tileset devant les yeux, vous voyez tous les tiles.
Cliquez sur un tile pour le sélectionner, vous reviendrez automatiquement à la partie "level"
Si vous maintenez la touche CTRL de gauche, vous passez en mode "mur". Vous pouvez alors sélectionner, pour chaque tile, s'il est un mur ou non.



Tout mur apparaîtra en violet mauve.

Partie "level"



Utilisez la souris pour peindre le level avec le tile sélectionné par la partie "choix du tile", alternez entre les deux parties pour changer de tile.
Utilisez la molette pour agrandir ou rétrécir le monde. Utilisez les touches H et V pour sélectionner un agrandissement horizontal ou vertical.
Utilisez les flèches pour faire défiler la carte si vous avez agrandit le monde plus que la taille de la fenêtre par défaut.
Utilisez CTRL+S pour sauvegarder le travail. Si vous avez démarré avec un .bmp, le même nom de fichier sera créé avec comme extension .txt.

Ce petit éditeur vous épargnera le fait de remplir vos niveaux à la main.

Nous savons maintenant comment mettre en place une carte avec possibilité de scrolling. Par la suite, nous allons voir comment mettre un personnage sur cette carte, et le faire évoluer.

Simple personnage

> Nous allons voir comment est défini un personnage, comment simplifier sa gestion, et voir comment l'insérer dans un monde simple.

Qu'est-ce qu'un personnage ?

Avant de se lancer dans du code, essayons de définir ce qu'est un personnage dans un petit jeu de plateforme, ou un jeu vu de dessus.

Il est important de définir quelques notions avant d'aller plus loin. Dans un jeu comme Mario Bros, Mario est dans le monde, et il **bouge**.

Ce mot **bouge** n'est pas précis. En réalité, le personnage bouge de deux manières indépendantes : l'animation, et le déplacement.

L'animation

Pour bien comprendre ce qu'on appellera l'animation, prenons n'importe quel GIF animé du net, sans couleur transparente derrière :

Un très joli petit Mario animé...

> *Que remarque-t-on ?*

Nous remarquons que le Mario "bouge" à l'intérieur d'un carré vert (qui est sa boîte englobante), qui, elle, **reste fixe**. J'ai désactivé la transparence du gif pour bien voir cette boîte verte.

Ce qui se passe à l'intérieur de cette boîte verte est **l'animation**.
Ce sont les pieds de Mario qui bougent, c'est un joli dessin qui s'anime...

Le déplacement

Dans un jeu, l'animation ne suffit pas, il faut qu'on puisse déplacer notre personnage. Concrètement, il faut faire bouger notre boîte verte, la faire avancer dans le monde, tout simplement.

> Il est important de faire la différence entre déplacement et animation !

Tous les personnages des jeux dont nous parlons sont animés, et se déplacent.
Mario en fait partie.

- Parfois, l'animation est réduite à une seule image, donc il n'y a finalement pas d'animation. C'est le cas d'un nuage qui se déplace, d'un missile qui se déplace. Cela est de plus en plus rare, car on animera le missile (en le faisant tourner, en animant le réacteur...) pendant son déplacement pour un meilleur esthétisme. Qui a dit qu'un missile ne pouvait pas être esthétique ? :-°
- Parfois, le sprite ne bouge pas, mais s'anime : c'est le cas de tous les gifs animés du net par exemple. C'est le cas d'un Ryu qui danse en garde avant le *FIGHT !*

Première approche de collision avec le décor

Nous allons maintenant parler collisions.

Notre but va être le suivant : Mario va se déplacer dans un monde avec des murs.

> Il ne faut pas que Mario passe à travers les murs...

Reprenons notre petit Mario animé :

> *Comment savoir s'il touche un mur ?*

Voici deux solutions.

Le pixel perfect

Le pixel perfect est un algorithme de collision perfectionniste qui va dire :

> Si un seul pixel de Mario touche un mur, alors il y a collision.

Nous oublions donc notre boite englobante verte pour cet algorithme :

Il va falloir déterminer, pour chaque pixel, s'il touche ou non un mur.
Sachant que notre Mario est animé, il se peut qu'à une frame d'animation, il ne touche pas le mur, et qu'à une autre, il le touche.
Que faisons nous alors ? On peut le faire reculer... Du coup, s'il continue d'être animé, et qu'on le fait déplacer vers le mur, on le verra trembler, car à chaque frame de l'animation, il sera déplacé. Disons le tout de suite, ça sera moche.

- Ce sera moche !
- Ce sera calculatoire, car il faudra déterminer s'il y a collision pour chaque pixel. Notre Mario est petit, mais s'il était plus grand, le nombre de calculs exploserait...
- On aura des problèmes de collisions qui dépendront de la frame d'animation en cours.

Ce sont quelques problèmes que peut soulever le *pixel perfect*.

Il pourrait en poser encore bien d'autres : dans un jeu du genre Zelda, si le pixel perfect était appliqué, on pourrait coincer notre bouclier entre deux branches d'arbre du décor. Pour s'en sortir, ce ne serait pas simple.
Il y a d'autres problèmes qui pourraient arriver. Le pixel perfect est - selon moi - un nid à problèmes. Évidemment, cela n'engage que moi. Il peut être utile dans certains cas, mais sûrement pas dans notre cas à nous.

Nous allons donc oublier cet algorithme pour notre sujet, il n'est pas adapté.

Collision par boite englobante (AABB)

Une *Axes Aligned Bounding Box* (AABB) est le nom qu'on donne à la boite englobante verte de Mario :

On la définit par son origine (le point en haut à gauche), sa largeur et sa hauteur (d'ailleurs, SDL_Rect est typiquement fait pour ça).
Elle est alignée avec les axes du monde : pas de losanges, mais un beau rectangle "droit".

On partira du principe que si la boite englobante touche un mur, alors Mario touche. Et ceci indépendamment de la frame d'animation de ce dernier. Si la boite touche, ça touche, et si elle ne touche pas, alors Mario ne touche pas.

Observez bien cette boîte englobante, elle est suffisamment serrée autour de Mario pour que cette gestion des collisions soit suffisante.

L'algorithme de collision avec le décor que nous allons voir rapidement ici ne considèrera **que** la boite englobante.

> On oublie donc l'animation de Mario, on ne parle plus que de la boîte !

Voici donc notre boîte verte :

Précaution

Pour utiliser cet algo, et donc faire abstraction des frames d'animation pour les collisions, il y a une précaution à prendre au niveau de l'animation.

L'animation est définie par un ensemble de petits dessins qu'on blit à la coordonnée (x,y) voulue, on en colle une à l'autre en fonction du temps passé. C'est ça qui fait l'animation.
Il est **fondamental** que ces petits dessins fassent tous la même taille (la même hauteur, la même largeur) : ainsi, la taille de la boîte englobante reste invariante d'une frame à l'autre.
C'est le cas de notre petit Mario. Vous pouvez constater sur le petit gif animé que si Mario bouge, la boîte verte, elle, reste fixe.

Sans cette contrainte, la boîte englobante se déformerait lors de l'animation, et, selon la frame, le rectangle, sans se déplacer, pourrait être dans le mur pour une frame, hors du mur pour une autre.
En garantissant les images de la même taille pour chaque frame, ce problème n'existe plus : soit le personnage est dans un mur, soit il ne l'est pas, et cela pour toutes ses animations.

Principe dans un monde

Notre boîte verte se déplace dans un monde, il faut simplement empêcher qu'elle rentre dans un mur.
Le principe est simple.

Nous définissons une fonction fondamentale, qu'on appellera **CollisionDecor**.
Cette fonction prend la boîte verte (sa position, sa largeur, sa hauteur), et nous dit simplement :

- tu es dans un mur ;
- tu n'es pas dans un mur.

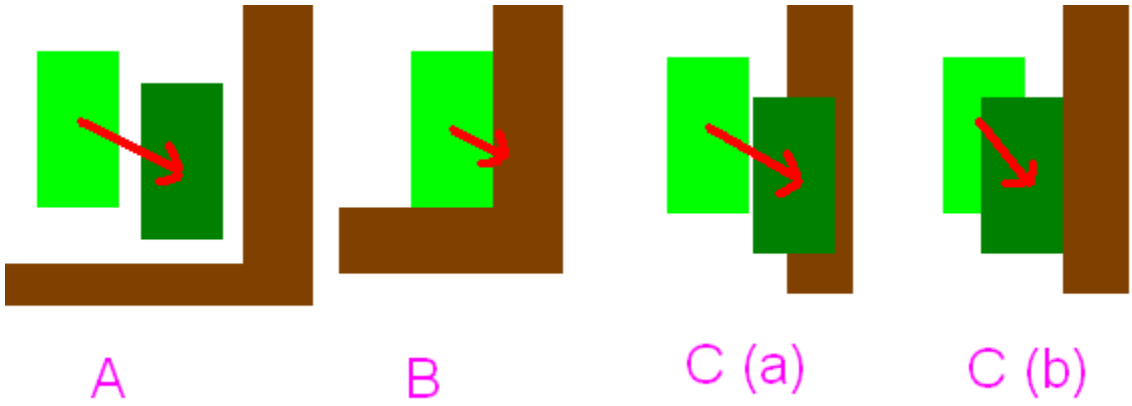
Informatiquement parlant, elle renvoie 1 si on touche un mur, 0 sinon.

> Cette fonction clé va faire le pont entre le personnage et le décor.

Tel un aveugle qui se ballade dans la rue, on veut juste savoir, à tout moment, si on touche un mur ou non. A partir de cette seule fonction, on va mettre en place nos collisions.

Algorithme de déplacement

Nous souhaitons déplacer notre boîte verte dans le décor.
Dans cette partie, nous allons voir comment faire, en faisant intervenir notre fonction **CollisionDecor**.
Voici le schéma suivant :



Au départ, nous avons notre boîte verte claire, qui est hors mur. En effet, il est interdit d'être dans un mur. La position initiale doit donc être hors mur.

Nous allons déplacer notre boîte verte selon un vecteur de déplacement (qui est rouge sur l'image ci dessus). Le but est que le modèle se déplace, mais qu'à sa position finale, il soit toujours hors mur.

Voici une première version de l'algorithme.

- La boîte verte claire est a une position initiale hors mur, nous donnons le vecteur de déplacement souhaité.
- Nous calculons l'éventuelle position finale (verte foncée).
- Nous demandons à CollisionDecor si la boîte verte foncée est dans le mur ou non.
- Si elle ne l'est pas, on valide le déplacement : notre boîte verte claire prend la place de la boîte verte foncée (cas A).
- Sinon, nous ignorons le déplacement, on ne bouge pas notre boîte verte (cas B).
- Dans tous les cas, nous sommes toujours hors mur à ce moment là.

Avec cet algorithme, le cas A et le cas B fonctionnent : on se déplace si on peut, on ne bouge pas si on ne peut pas.

Le cas C

Le cas C est plus complexe. Notre rectangle vert clair ne touche pas le mur, mais n'est pas collé contre non plus. Si on souhaite le faire bouger selon le vecteur rouge, la nouvelle position calculée rentrera dans le mur (cas C(a)), ce n'est pas bon, donc ce n'est pas validé. L'algorithme ci-dessus ne fera donc pas bouger du tout le rectangle vert clair -> nous nous retrouverons dans le cas B, sauf que nous ne sommes pas collés au mur.

> Notre rectangle ne touchera donc jamais le mur, il restera toujours à quelques pixels de celui-ci. C'est gênant.

L'idée est que si le mouvement nous amène dans le mur, de voir si on ne pourrait pas modifier le vecteur rouge de façon à faire un plus petit mouvement pour s'en approcher au plus près. Le mieux étant d'aller le toucher, se coller à lui au pixel près, mais sans qu'il y ait collision. Nous aurons alors le cas C(b). J'appellerai cette opération **Affiner le mouvement**.

Voici donc une deuxième version de l'algorithme.

- La boîte verte claire est à une position initiale hors mur, nous donnons le vecteur de déplacement souhaité.
- Nous calculons l'éventuelle position finale (verte foncée).
- Nous demandons à CollisionDecor si la boîte verte foncée est dans le mur ou non.
- Si elle ne l'est pas, on valide le déplacement : notre boîte verte claire prend la place de la boîte verte foncée (cas A).
- Sinon, nous sommes dans le cas B ou le cas C.
    - Nous cherchons un vecteur affiné, qui permettrait d'aller se coller contre le mur, en faisant des essais, et en retestant avec CollisionDecor.

    - Si ce vecteur est nul (on est déjà collé sur le mur) alors on ne bouge pas (cas B).
    - Sinon on se déplace de ce vecteur, et on se retrouve collé au mur (sans rentrer dedans bien sûr !) et on valide le déplacement (cas C(b)).
- Dans tous les cas, nous sommes toujours hors mur à ce moment là.

Nous proposerons plus loin des méthodes pour affiner.

Code exemple

Maintenant que nous avons vu le principe que nous allons mettre en place, voici un code qui va illustrer tout cela, dans un monde très très simple pour commencer !
Ce monde sera juste un rectangle marron...

Prenez le projet prog4, et compilez le, et lancez le.

> Utilisez les flèches pour déplacer le rectangle vert, echap pour quitter.

> Vous constatez rapidement que quand on s'approche du rectangle marron, il y a quelques soucis. Lisez bien cette partie jusqu'au bout pour résoudre cela.

Si on regarde le main, dans prog4.c :

```
#include "fevent.h"
#include "fsprite.h"

void RecupererVecteur(Input* in,int* vx,int* vy)
{
        int vitesse = 5;
        *vx = *vy = 0;
        if (in->key[SDLK_UP])
                *vy = -vitesse;
        if (in->key[SDLK_DOWN])
                *vy = vitesse;
        if (in->key[SDLK_LEFT])
                *vx = -vitesse;
        if (in->key[SDLK_RIGHT])
                *vx = vitesse;
}

void Evolue(Input* in,SDL_Rect* mur,Sprite* perso)
{
        int vx,vy;
        RecupererVecteur(in,&vx,&vy);
        DeplaceSprite(perso,mur,vx,vy);
}

int main(int argc,char** argv)
{
        SDL_Rect mur;
        Sprite* perso;
        SDL_Surface* screen;
        Input in;
        memset(&in,0,sizeof(in));
        SDL_Init(SDL_INIT_VIDEO);               // prepare SDL
        screen = SDL_SetVideoMode(800,600,32,SDL_HWSURFACE|SDL_DOUBLEBUF);
        mur.x = 450;
        mur.y = 100;
        mur.w = 100;
        mur.h = 200;
        perso = InitialiserSprite(101,150,50,100);
        while(!in.key[SDLK_ESCAPE])
        {
                UpdateEvents(&in);
                Evolue(&in,&mur,perso);
                SDL_FillRect(screen,NULL,0);  // nettoie l'ecran en noir
                SDL_FillRect(screen,&mur,0x800000);  // affiche le mur
                AfficherSprite(perso,screen);
                SDL_Flip(screen);
                SDL_Delay(5);
        }
        LibereSprite(perso);
        SDL_Quit();
        return 0;
}
```

On voit dans le main que je crée un SDL_Rect, que j'appelle mur. Ce sera mon décor. Je lui donne une position, et des dimensions.
Puis j'initialise un sprite, c'est à dire un objet mobile. Nous verrons plus loin par quoi est défini ici un sprite pour le moment.

dans la boucle, on "Evolue", puis on affiche le mur, et le sprite.

Evolue

La fonction Evolue, juste au dessus, fait 2 choses : d'abord, elle récupère un vecteur de déplacement via la fonction RecupererVecteur qui est au dessus, puis elle déplace le sprite selon ce vecteur.

RecupererVecteur

Cette fonction est très simple, elle lit les touches du clavier (les flèches) et met un vecteur à jour. 8 positions possibles (les 4 directions ainsi que les diagonales) ainsi que le vecteur nul possible.

Passons maintenant au fichier fsprite.h

```
#include <sdl/sdl.h>

#pragma comment (lib,"sdl.lib")      // ignorez ces lignes si vous ne linkez pas les libs de cette f
açon.
#pragma comment (lib,"sdlmain.lib")

typedef struct
{
        SDL_Rect position;
} Sprite;

Sprite* InitialiserSprite(Sint16 x,Sint16 y,Sint16 w,Sint16 h);
void LibereSprite(Sprite*);
int DeplaceSprite(Sprite* perso,SDL_Rect* mur,int vx,int vy);
void AfficherSprite(Sprite* perso,SDL_Surface* screen);
```

Vous voyez qu'actuellement, un sprite, ce n'est qu'un SDL_Rect.
On peut l'initialiser, le libérer quand on a fini, puis le déplacer, et l'afficher.

Passons a fsprite.c

```
#include "fsprite.h"

#define SGN(X) (((X)==0)?(0):(((X)<0)?(-1):(1)))
#define ABS(X) ((((X)<0)?(-(X)):(X)))

Sprite* InitialiserSprite(Sint16 x,Sint16 y,Sint16 w,Sint16 h)
{
        Sprite* sp = malloc(sizeof(Sprite));
        sp->position.x = x;
        sp->position.y = y;
        sp->position.w = w;
        sp->position.h = h;
        return sp;
}
void LibereSprite(Sprite* sp)
{
        free(sp);
}

int CollisionDecor(SDL_Rect* m,SDL_Rect* n)
{
        if((m->x >= n->x + n->w)
                || (m->x + m->w <= n->x)
                || (m->y >= n->y + n->h)
                || (m->y + m->h <= n->y)
                )
                return 0;
        return 1;
}

int EssaiDeplacement(Sprite* perso,SDL_Rect* mur,int vx,int vy)
{
        SDL_Rect test;
        test = perso->position;
        test.x+=vx;
        test.y+=vy;
        if (CollisionDecor(mur,&test)==0)
        {
                perso->position = test;
                return 1;
        }
        return 0;
}

void Affine(Sprite* perso,SDL_Rect* mur,int vx,int vy)
{
        int i;
        for(i=0;i<ABS(vx);i++)
        {
                if (EssaiDeplacement(perso,mur,SGN(vx),0)==0)
                        break;
        }
        for(i=0;i<ABS(vy);i++)
```

```
        {
                if (EssaiDeplacement(perso,mur,0,SGN(vy))==0)
                        break;
        }
}

int DeplaceSprite(Sprite* perso,SDL_Rect* mur,int vx,int vy)
{
        if (EssaiDeplacement(perso,mur,vx,vy)==1)
                return 1;
        /*Affine(mur,perso,vx,vy);*/
        return 2;
}

void AfficherSprite(Sprite* perso,SDL_Surface* screen)
{
        SDL_Rect copyperso;
        copyperso = perso->position;
        SDL_FillRect(screen,&copyperso,0x00FF00);  // affiche le perso
}
```

Regardons tout d'abord les fonctions les plus simples :
InitialiserSprite,LibereSprite ne devraient pas poser de soucis.

AfficherSprite contient une légère astuce : au lieu de passer directement le SDL_Rect du sprite à la fonction SDL_FillRect, je passe une copie.
Tout comme SDL_BlitSurface, si vous ne passez pas une copie, vous risquez d'avoir des problèmes si vous faites sortir votre sprite à gauche ou en haut de l'écran. Passer une copie permet de ne pas avoir ce problème.

DeplaceSprite

Nous voila à la fonction la plus complexe de ce programme.

Tout d'abord, la fonction lance EssaiDeplacement. Si cet essai est bon, on sort de la fonction. Sinon, on ne fait rien car la suite est commentée.

EssaiDeplacement

La fonction EssaiDeplacement va essayer de déplacer le sprite selon le vecteur donné. Je dis essayer, car elle prend le décor en paramètres (ici un simple mur), et si le déplacement nous amène dans un mur, elle ne déplace pas et renvoie 0, comme on a vu plus haut dans le cas B, ou le cas C(a).
Si elle arrive à nous déplacer (cas A), elle met à jour perso->position et retourne 1 pour dire qu'elle a réussi.

CollisionDecor

La fonction CollisionDecor est l'algorithme de collision AABB que je détaille ici (http://www.siteduzero.com/tutoriel-3-254500-formes-simples.html#ss_part_2).
Elle renvoie 1 si et seulement si les deux rectangles se chevauchent.

> Arrêtons de détailler le code ici pour le moment, et relançons le programme

Allez vous coller contre le rectangle marron. Vous pouvez constater que si vous arrivez par la gauche, vous ne pouvez pas vous coller au pixel près.
Pire, lorsque vous êtes presque collé, si vous appuyez en même temps à droite et en haut, votre sprite ignore l'appui sur le haut, alors que vous pourriez monter.
Dans ce dernier cas, vous êtes dans le cas C(a) et il n'y a pas d'affinage.

Reprenons le code. Regardez la fonction DeplaceSprite.
Vous pouvez constater qu'une ligne est commentée :

```
/*Affine(mur,perso,vx,vy);*/
```

Décommentez les et relancez le programme.
Oh miracle, vous pouvez maintenant aller vous coller au pixel près, et glisser sur le mur.

La fonction Affine

La fonction affine va juste prendre le vecteur, et tenter de s'approcher pixel par pixel en X d'abord, puis en Y ensuite.
La macro ABS renvoie la valeur absolue de la valeur passée
la macro SGN renvoie 1 si la valeur est positive, -1 si elle est négative (0 si elle est nulle)

La fonction Affine va faire des essais pixel par pixel jusqu'à aller se coller contre.
C'est le cas C(b).

A la fin de cette première partie, nous avons fait les premiers pas vers la collision avec un décor. Ce décor-ci était très simple. Cependant, nous verrons que même si le décor est complexe, même s'il y a scrolling, et même si nous ne contrôlons pas directement le vecteur de déplacement lorsque c'est une fonction de gestion de physique qui le fait, le concept restera le même.

Nous allons progressivement parler de tout ça, en ajoutant, étape par étape, les nouveaux éléments.

Insertion dans un monde de Tiles

Nous avons vu dans la partie 1 comment créer un monde fait de tiles.
Nous avons vu au début de la partie 2 comment gérer un personnage simpliste et ses collisions avec un mur simple.
Voyons maintenant comment relier le tout !

La nouvelle fonction CollisionDecor

Comme je vous le disais en conclusion de la sous-partie précédente, le concept de déplacement va rester le même : la fonction *DeplaceSprite* et ses sous-fonctions vont ressembler à celles d'avant, le majeur changement va être le changement de la fonction CollisionDecor, celle qui dit "tu es dans un mur ou pas".

Nous allons donc voir comment cette fonction peut marcher pour un monde fait de tiles.

Rappelons que nous raisonnons dans le "grand monde", qui, même s'il n'est pas affiché, est calculable.
C'est à dire que si je suis dans un grand monde, mon personnage peut très bien être à la position x = 10000, y = 8625 par exemple...

Dans le chapitre d'avant, cette fonction se ramenait à simplement tester une collision entre deux rectangles : le personnage et le mur.

Voyons pour tester les collisions dans un monde de tiles.

La solution violente

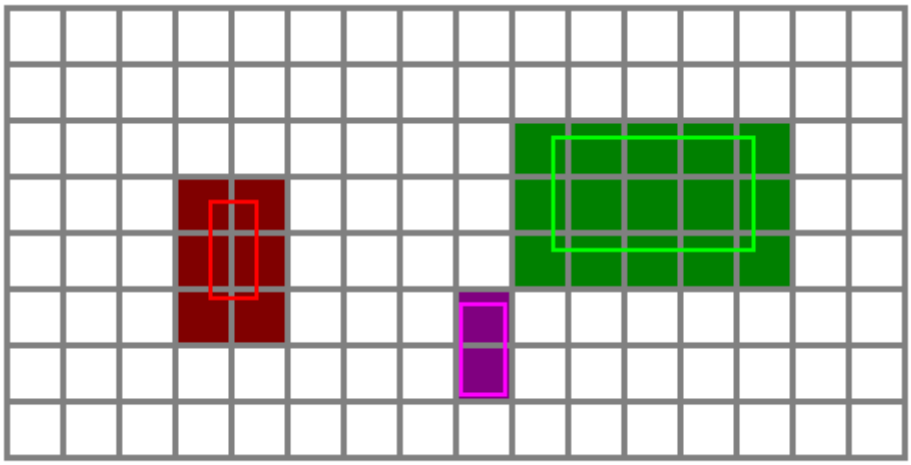
La première idée est de se dire que dans l'exemple d'avant, j'avais un mur, je testais avec un algorithme de collision boite/boite.
Ici, dans mon monde, j'ai davantage de murs, je teste avec chacun, et si on en touche un, on renvoie 1.

Le gros problème qu'on voit tout de suite est que, dans un monde de Mario, on est au début du stage, et on va tester tous les murs du stage, y compris les blocs de l'arrivée, qui sont loin ! Loin et nombreux !

C'est très calculatoire, beaucoup trop violent à infliger à sa machine, et à oublier rapidement.

Localiser les tiles qui pourraient intervenir

L'idée première va être de localiser les tiles concernés. C'est-à-dire que si notre Mario est au début du stage, inutile de tester les tiles de la fin du stage : on ne va tester que ceux qu'il touche.



Ci-dessus un petit dessin, nous voyons le monde complet, découpé en tiles (en gris), et plusieurs personnages schématisés par leur boîte englobante (notre fameuse boîte verte).
En clair derrière eux, les tiles que le perso touche.

> Tester ces tiles-la, et uniquement ceux-la, est suffisant.

L'algorithme va être le suivant :

- Localiser les tiles concernés (les tiles colorés) en fonction du personnage.
- Tester chacun de ces tiles : si l'un d'entre eux est un mur, on renvoie 1 (collision).
- Sinon, on renvoie 0 (pas de collision).

Localiser les tiles concernés

L'ensemble des tiles concernés formera un rectangle. Ce rectangle aura comme premier tile celui d'en haut à gauche, le tile de coordonnées (xmin,ymin), et comme dernier tile celui d'en bas à droite, le tile de coordonnées (xmax,ymax).

Déterminer ces quatre données sera très rapide : pour (xmin,ymin), il va falloir déterminer dans quel tile est le point en haut à gauche de notre rectangle de personnage. Pour déterminer (xmax,ymax), il va falloir déterminer dans quel tile est le point en bas à droite de notre rectangle de personnage.
Regardez de nouveau le dessin, la règle est respectée.

Déterminer dans quel tile est un point est extrêmement rapide : notre monde est régulier, et le premier tile commence à la coordonnée (0,0).
Pour un personnage ayant comme boîte x,y (point en haut à gauche personnage), et largeur w et hauteur h, on peut écrire :

xmin = x/LARGEURTILEymin = y/HAUTEURTILE

Le point en bas à droite du personnage est calculé à partir de ses coordonnées (x,y) auxquelles on ajoute respectivement w-1 et h-1.

x_{basdroite} = x + w - 1y_{basdroite} = y + h - 1

De ce fait, nous avons :

xmax = x_{basdroite}/LARGEUR_TILEymax = y_{basdroite}/HAUTEUR_TILE

> Le calcul de ces valeurs ne dépend absolument pas de la taille du monde : de ce fait, même si le monde est immense, le calcul ne sera pas plus long.

Boucle de tests à faire

Une fois qu'on a déterminé le rectangle de tiles à tester, il faut tous les tester. Nous n'échapperons pas à un double for :

```
int i,j;
for(i=xmin;i<=xmax;i++)
{
  for(j=ymin;j<=ymax;j++)
  {
     // tester un tile, si on touche un mur, inutile d'aller plus loin : on retourne 1
  }
}
```

La vitesse de cet algo dépend directement de la taille de votre personnage. Si votre personnage est petit, on testera 2, voir 4, voir 6 tiles (tout dépend du chevauchement de tiles de votre personnage). Ce n'est pas fixe, regardez le dessin ci-dessus, le rectangle violet et le rouge font à peu près la même taille, mais leur position n'est pas la même, et le rouge est à cheval sur davantage de tiles.

Le carré vert est un plus gros personnage (un gros monstre par exemple), on testera donc davantage de tiles pour lui.

> Notre algorithme de collision fonctionnera sur tout type de personnages, de taille quelconque.

Même si ce double for dépend de la taille du personnage, il sera rapide, car à moins de faire des super-monstres-énormes, il y aura toujours peu de tiles à tester !

Pour les furieux de l'optimisation, on peut toujours aller plus loin :
Au lieu de tester tous les tiles entre xmin,ymin / xmax,ymax, on ne peut tester que ceux du bord du rectangle, et pas les tiles intérieurs. En effet, on peut partir du principe qu'un gros personnage est hors de tout mur, et qu'il ne pourra jamais avoir de mur à l'intérieur de lui, car les tests de collision avec les bords auront empêché ça. De ce fait, seuls les tiles du bord du rectangle de tiles concernés peuvent être testés. Sur le dessin ci-dessus, on pourrait ne pas tester les 3 tiles au milieu du rectangle de tiles vert.
Cette optimisation n'a de raison d'être que pour les très gros personnages, pas pour un Mario...
Souvent dans les jeux, les gros monstres sont d'ailleurs dans des zones ouvertes, et on ne teste pas leur collision avec le décor.

Tester un tile (version lente)

Dans la boucle qu'on aura faite, il faut donc tester un tile. La version lente consiste à récupérer le tile à tester, à récupérer sa boîte englobante, et lancer un algo de collision boîte/boîte (vu au chapitre précédent) pour voir si on touche.
Même si c'est rapide, c'est un peu bête car le travail est pré-mâché pour aller bien plus vite.

Tester un tile (version optimale)

Nous testons un tile à la position i,j. Si nous le testons, c'est que le perso le chevauche. Il suffit juste de voir si ce tile est identifié comme un "mur" ou pas. C'est tout !

La fonction complète

Voici donc la fonction complète qui résume tout ce que nous avons vu :

```
int CollisionDecor(Sprite* perso)
{
        int xmin,xmax,ymin,ymax,i,j,indicetile;
        Map* m = perso->m;
        if (perso->x<0 || (perso->x + perso->w -1)>=m->nbtiles_largeur_monde*m->LARGEUR_TILE
         || perso->y<0 || (perso->y + perso->h -1)>=m->nbtiles_hauteur_monde*m->HAUTEUR_TILE)
                return 1;
        xmin = perso->x / m->LARGEUR_TILE;
        ymin = perso->y / m->HAUTEUR_TILE;
        xmax = (perso->x + perso->w -1) / m->LARGEUR_TILE;
        ymax = (perso->y + perso->h -1) / m->HAUTEUR_TILE;
        for(i=xmin;i<=xmax;i++)
        {
                for(j=ymin;j<=ymax;j++)
                {
                        indicetile = m->schema[i][j];
                        if (m->props[indicetile].plein)
                                return 1;
                }
        }
        return 0;
}
```

On retrouve xmin,xmax,ymin,ymax calculés comme nous avons vu.
Il y a en dessous un petit test, qui regarde si le perso sort du monde. Nous partons du principe que le perso ne doit pas sortir du monde. Donc si une de nos quatre valeurs est hors du monde, en renvoie 1 -> on touche. Concrètement, tout se passe comme si le monde était entouré par un mur. Vous verrez dans l'exemple ci dessous qu'on ne peut pas sortir.

> On pourrait changer ça, et dire que le bord du monde ne contient pas de mur. Dans ce cas, il suffirait de ramener les coordonnées xmin,ymin / xmax,ymax dans le monde. On ne testerait ainsi que les rectangles du monde.

> Quoi que vous choisissiez comme principe quand le perso sort du monde, il faut faire attention à ce que xmin,ymin / xmax,ymax soient des valeurs dans le monde. En effet, ces valeurs vont être utilisées pour tester le tableau carte->schema[][] de la structure Map. Il ne faut pas déborder, sous peine de plantage...

On retrouve ensuite le double for. Dedans, pour chaque i,j concerné, on regarde l'indice du tile concerné, et on va voir dans le tableau props si ce tile a la propriété mur activée. Si c'est le cas, on renvoie qu'on touche, sans même avoir besoin de finir le for.

Si on sort du for, c'est qu'aucun tile mur n'a été touché, alors on renvoie 0 -> on ne touche pas.

> Cette fonction est peu calculatoire, très rapide, et fiable. Tant mieux car c'est une fonction appelée souvent, il faut qu'elle soit rapide. C'est le cas !

Les déplacements rapides

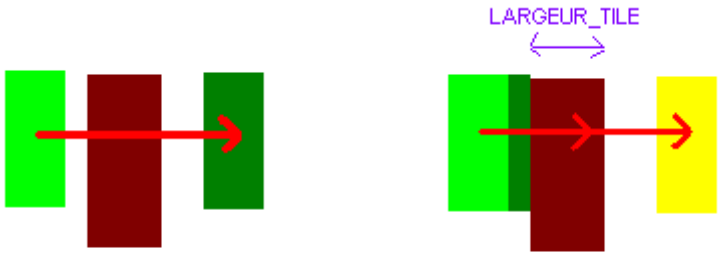
Avant de voir le code, et un beau petit exemple de collision dans notre petit monde, soulevons un petit problème.

Nous avons vu comment nous faisons nos déplacements : on applique un vecteur de translation, si la position finale est valide, on bouge, sinon, on affine ou on ne bouge pas.

Cela marche très bien, mais à une seule condition : qu'on ne bouge pas trop vite.

Imaginons un Sonic qui fonce. Son vecteur de déplacement devient grand, il *tabule*, c'est à dire qu'à une frame on le dessine à une position, à une autre, on le dessine beaucoup plus loin (en réalité, avec le scrolling, c'est la carte qui tabule, mais ça revient au même).

Voici le problème que ça peut poser :



A gauche, on voit notre boîte verte. On veut la déplacer selon le vecteur rouge. Or, il y a un mur. Mais le vecteur rouge est grand, donc l'algorithme qu'on a vu plus haut va tenter de le déplacer, va réussir, car la position test est hors mur. Et Sonic aura traversé le mur... C'est moche, non ?

Il y a danger que cela arrive si, pour un vecteur vx,vy, vx>= LARGEUR_TILE ou vy>=HAUTEUR_TILE.
Par contre, si cette condition n'est pas remplie, aucune chance d'avoir ce problème.

Il faut donc éviter ce cas :

- soit en empêchant de se déplacer trop vite ;
- soit en coupant le déplacement en plusieurs morceaux acceptables.

Nous n'allons bien sûr pas vous empêcher d'aller vite. Nous allons voir comment faire pour couper le déplacement en deux.
Si vous regardez la partie droite du dessin, vous voyez qu'au lieu de translater d'un grand vecteur rouge, je translate deux fois d'un plus petit vecteur rouge. Et ce vecteur nous permettra de bien se payer le mur.
On va lancer deux fois la fonction Deplace, avec un vecteur réduit de moitié à chaque fois.
La première fois, on va aller cogner le mur, et l'affinage va nous plaquer contre.
La deuxième fois, à partir de la position collée contre le mur, on n'avancera pas.
Le problème disparait.

L'algorithme qu'on va mettre en place dans la fonction Deplace est simple et récursif :

```
int DeplaceSprite(Sprite* perso,int vx,int vy)
{
        if (vx>=perso->m->LARGEUR_TILE || vy>=perso->m->HAUTEUR_TILE)
        {
                DeplaceSprite(perso,vx/2,vy/2);
                DeplaceSprite(perso,vx-vx/2,vy-vy/2);
                return 3;
        }
        if (EssaiDeplacement(perso,vx,vy)==1)
                return 1;
        Affine(perso,vx,vy);
        return 2;
}
```

Nous voyons que si vx ou vy sont trop grand, on relance deux fois la fonction avec un vecteur deux fois plus petit dans un premier temps, suivi du "reste", donc également un vecteur deux fois plus petit.

L'avantage de la récursivité, c'est que si les nouveaux vecteurs sont toujours trop grand, on redécoupe, quitte à relancer l'algo quatre fois, huit fois... Jusqu'à ce que le vecteur soit acceptable !

Si ce chapitre vous échappe (car vous n'aimez pas la récursivité ou ne comprenez pas tout) ignorez, mais faites attention à vos vitesses... Ou alors acceptez la fonction telle qu'elle est.

> Notez qu'avec des nombres entiers, vx-vx/2 n'est pas forcément égal à vx/2

Code exemple

Nous finirons ce chapitre par du code qui reprend ce que nous avons vu plus haut.

Prenez le programme "prog5".

Compilez le et lancez le. Utilisez les flèches pour déplacer le rectangle vert, et les touches "fgth" pour contrôler le scrolling.
Vous pouvez remarquer que vous ne pouvez pas rentrer dans les décors.
Sauf les barrières et fleurs, car elles sont renseignées comme "vide".

Vous ne pouvez pas sortir non plus du monde, car la fonction CollisionDecor renvoie qu'il y a collision si on sort.

Voyons un petit peu le code.
Beaucoup de choses réutilisées des anciens codes.

Ici, j'ai changé la structure Sprite dans fsprite.h

```
typedef struct
{
        Map* m;
        int x,y,w,h;
} Sprite;
```

Les sprites embarquent maintenant un pointeur vers la map. Et les x,y,w,h ne sont plus dans SDL_Rect, parce que les x,y pourront devenir très grands. En effet, les coordonnées du sprite seront celles du repère global.
Cela pourra permettre au sprite de pouvoir sortir de la zone de scrolling, en continuant d'être actif.

L'intérêt de stocker les coordonnées globales, c'est que le scrolling le déplacera automatiquement, il n'y aura pas de mouvement à compenser par le scrolling.

La fonction AfficherSprite

```
void AfficherSprite(Sprite* perso,SDL_Surface* screen)
{
        SDL_Rect R;
        R.x = perso->x - perso->m->xscroll;
        R.y = perso->y - perso->m->yscroll;
        R.w = perso->w;
        R.h = perso->h;
        SDL_FillRect(screen,&R,0x00FF00);  // affiche le perso
}
```

Elle tient compte des paramètres de fenêtrage qu'elle va chercher via le lien qu'embarque chaque sprite.
Notez bien que s'il y a plusieurs sprite, ils auront tous un lien vers la même map. Il n'y a pas une map par sprite.

Tout le reste est de la réutilisation des chapitres précédents.

Nous avons enfin intégré notre personnage (même réduit à un simple rectangle) dans le monde des tiles.
Il ne passe plus à travers les murs.

Pour l'instant, il est vert et carré, et il vole.
Notez que les jeux vus de dessus (comme Zelda Link to the past, ou la philosophie RPG Maker), utilisent ce concept.
Dans ces jeux, pas besoin de gravité.

Vous pouvez vous appuyer sur l'exemple en l'état pour faire un jeu vu de dessus, même s'il est préférable de continuer à lire pour voir les diverses techniques que je vais vous proposer ! ;)

Scrolling automatique

L'exemple précédent proposait de se déplacer dans un monde, avec scrolling manuel : c'est-à-dire que c'était à vous de faire défiler l'écran avec des touches.
Tout joueur que vous êtes, vous savez que dans les jeux, ça ne se passe pas comme ça, que ce n'est pas vous qui faites manuellement défiler l'écran. L'ordi le fait pour vous.

Nous allons voir dans ce chapitre comment faire en sorte que l'ordi le fasse pour vous, qu'il suive automatiquement votre personnage pour un meilleur confort de jeu.

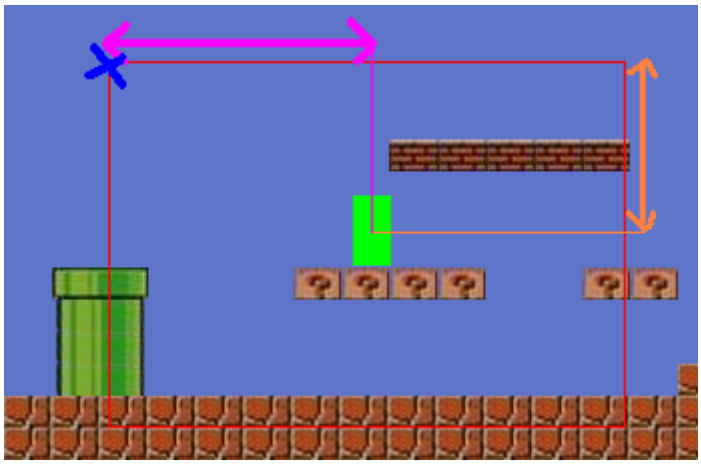
Je proposerai plusieurs méthodes.

Centre du monde !

Si vous aimez être le centre du monde, vous allez aimer cette méthode !

Avec cette méthode, nous souhaitons que le joueur soit toujours bien au milieu.
Nous allons donc ajuster le fenêtrage du scrolling pour faire en sorte que notre joueur soit au milieu. Nous déplaçons librement notre joueur, à l'ordi de nous suivre.

Si nous regardons le dessin ci-dessous :



Nous voyons le fenêtrage de scrolling en rouge. Nous voyons notre personnage bien centré au milieu du cadre rouge. En haut à gauche, le point bleu, c'est les coordonnées de nos paramètres xscroll et yscroll. C'est eux qu'il faut mettre à jour en fonction de la position de notre personnage.

C'est un peu de géométrie !

Quelques calculs

J'ai :

- xscroll, yscroll : coordonnées du point haut gauche de la fenêtre de scrolling.
- largeur_fenetre, hauteur_fenetre : largeur et hauteur de cette fenêtre.
- perso->x et perso->y : point haut gauche de mon perso (du rectangle vert).
- perso->w et perso->h : largeur et hauteur du perso.

On veut que le centre de la fenêtre corresponde au centre du perso.

Le centre du perso, c'est :
x = perso->x + perso->w/2y = perso->y + perso->h/2

Le centre de la fenêtre c'est :
x = xscroll + largeurfenetre/2y = yscroll + hauteurfenetre/2

On veut que ces centres coïncident, donc :
perso->x + perso->w/2 = xscroll + largeurfenetre/2perso->y + perso->h/2 = yscroll + hauteurfenetre/2

On cherche xscroll et yscroll, on résout, on trouve :
xscroll = perso->x + perso->w/2 - largeurfenetre/2yscroll = perso->y + perso->h/2 - hauteurfenetre/2

Et voilà !

Clamping

Nous avons besoin également de prendre une précaution.
Imaginons que notre perso soit très à gauche du monde. Si nous centrons la fenêtre de scrolling autour, alors une partie de cette fenêtre sortira du monde.
Avec les fonctions présentées depuis le début du tuto, il **ne faut pas** que cette fenêtre sorte du monde. Il faut qu'elle soit bloquée entièrement dans le monde.

Dans le jargon, si on empêche la fenêtre de sortir du monde, on dit qu'on la *clamp*.

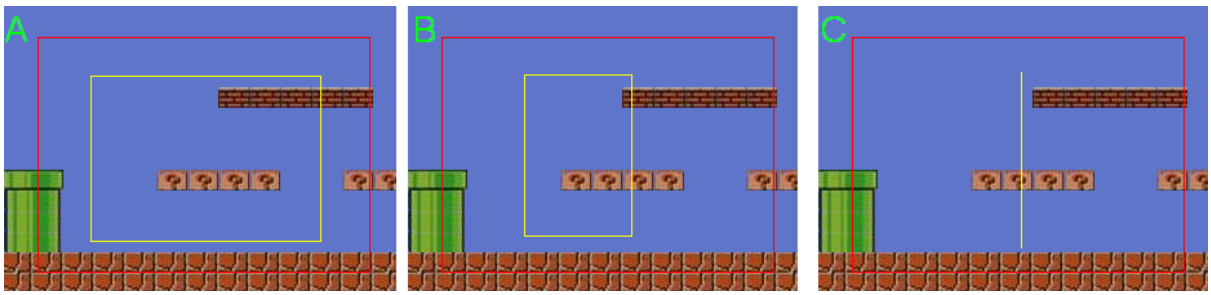
> Si jamais la fonction de *clamping* modifie xscroll et yscroll pour la remettre dans l'écran, on aura alors le seul cas ou notre personnage ne sera pas forcément au centre du monde, mais pourra aller se coller contre le bord. En effet, la fenêtre de scrolling s'arrête, mais vous, rien ne vous empêche de continuer à marcher jusqu'au bord, donc de voir réellement votre personnage se déplacer à droite dans votre fenêtre. Ceci se voit bien dans les jeux vidéos quand on approche du bord du stage, c'est le seul cas ou on peut se coller au bord de l'écran.

Cette méthode était donc à présenter pour notre culture, mais ce n'est pas celle que nous utiliserons à terme.
Essayez-la quand même dans le code d'en bas, pour voir.

Scrolling à sous-boîte limite

Dans la majorité des jeux de plateforme, ou vu de dessus, le personnage peut se déplacer dans l'écran. Le scrolling ne se met à le suivre que s'il va "trop au bord".

Voyons quelques schémas pour comprendre.



Nous reconnaissons le rectangle rouge : fenêtrage de scrolling, avec son point haut gauche qui est toujours xscroll et yscroll.

Nous définissons un rectangle jaune, à l'intérieur, que nous appellerons **rectangle limite**.

Le concept est simple : notre personnage peut se promener dans tout le rectangle jaune. Nous considérons son centre (au personnage). Si son centre est dans le rectangle limite, on ne scroll pas. Par contre, s'il en sort, on scroll de façon à le ramener dedans.

Il faudra alors calculer de combien on sort, en x et en y, et ajuster xscroll et yscroll de ces valeurs pour ramener notre personnage à la limite où il ne sort pas.

Avant de passer aux calculs, regardons notre dessin.

- La cas A est un rectangle jaune à égale distance dans les quatre directions au rectangle rouge. La conséquence, c'est que dès que le personnage arrive près du bord, il y a défilement. Cependant, si un ennemi arrive, comme on est près du bord, on aura peu de temps pour le voir.

- Le cas B est meilleur pour les jeux de plateforme où on avance à droite. Dès qu'on franchit le milieu, on défile. On voit ainsi bien les ennemis arriver. On peut revenir en arrière, et on ne défilera que si vraiment on va trop en arrière.
- Le cas C présente un rectangle réduit à une seule ligne. Conséquence, le personnage sera toujours au centre horizontalement, mais la caméra ne fera pas de bond avec le personnage au moment des sauts. Si on revient en arrière, la caméra suit immédiatement.

Vous pouvez ajuster votre rectangle limite comme vous le souhaitez pour avoir la meilleure ergonomie possible.

Notez que le cas limite où le rectangle est réduit à un seul point, au centre de l'écran, nous ramène dans le cas "centre du monde" ci-dessus.

Calculs

Voici les données que nous avons :

- xcsroll, yscroll : coordonnées haut gauche du fenêtrage de scrolling (rectangle rouge) ;
- largeurfenetre, hauteurfenetre : largeur et hauteur de fenêtre de scrolling ;
- xlim,ylim : coordonnées relatives du point haut gauche du rectangle jaune par rapport à xscroll et ysroll : distance fixe entre ces deux points ;
- wlim, hlim : largeur et hauteur de la fenêtre jaune. Plus petite que la hauteur et largeur de la fenêtre rouge ;
- xperso, yperso : coordonnées haut gauche perso ;
- wperso, hperso : largeur et hauteur perso.

Nous disions que si le centre du perso sort, alors on fait évoluer le scrolling. Donc tout ce qui compte pour le perso, c'est son centre.
Ce centre est défini par (cxperso,cyperso) de la façon suivante :
cxperso = xperso + wperso/2cyperso = yperso + hperso/2

Calculons les limites, à un instant donné, à ne pas dépasser. Cette limite évolue avec le scrolling.
Les coordonnées xlimmin et ylimin représenteront le coin haut gauche de la limite, et xlimmax et ylimmax le coin bas droit, dans le grand monde.

Pour le coin haut gauche, une simple addition suffit :
xlimmin = xscroll + xlimylimmin = yscroll + ylim

Pour le coin bas droit : c'est le coin haut gauche auquel on ajoute la largeur et la hauteur de la fenêtre de limite :
xlimmax = xlimmin + wlimylimmax = ylimmin + hlim

> Il suffit maintenant de vérifier si les coordonnées de notre centre ne sortent pas de ces limites. Et si c'est le cas, on ajuste. If suffira de 4 if, un pour chacune des 4 directions.

```
if (cxperso<xlimmin)
  xscroll -= (xlimmin-cxperso);
if (cyperso<ylimmin)
  yscroll -= (ylimmin-cyperso);
if (cxperso>xlimmax)
  xscroll += (cxperso-xlimmax);
if (cyperso>ylimmax)
  yscroll += (cyperso-ylimmax);
```

Une fois cette opération faite, on aura la garantie que notre sprite se trouve bien dans la zone jaune.
Cependant, si jamais notre fenêtre de scrolling est calculée hors du monde, il faut la *clamper* comme on a vu au dessus, et là, notre perso pourra donc se coller au bord.

Code exemple

Voici maintenant l'heure de voir le code qui illustre ce que nous venons de voir.

Ouvrez et compilez le projet "prog6".

Cet exemple ressemble beaucoup à celui de la partie précédente : vous allez déplacer votre personnage vert dans le même monde, à la différence près que la caméra vous suivra.
Utilisez donc uniquement les touches de direction du clavier pour tester cet exemple.

Nous avons vu 2 types de scrolling, l'exemple illustre le second.
Nous avions vu que le premier cas pouvait être considéré à partir du premier cas avec un rectangle limite ramené à un point central.
Vous pourrez modifier les paramètres de la fonction FocusScrollBox appelée dans le main pour l'envisager.

Vous pouvez remarquer que j'ai modifié légèrement les fichiers fsprite.h et fmap.h par rapport à l'exemple précédent.

Dans la mesure ou Map aura besoin de Sprite, et Sprite aura besoin de Map, nous pouvons voir :

- en haut de fmap.h

```
typedef struct Ssprite Sprite;
```

- en haut de fsprite.h

```
typedef struct Smap Map;
```

Regardons le main, dans le fichier prog6.c

```
int main(int argc,char** argv)
{
        Sprite* perso;
        SDL_Surface* screen;
        Map* carte;
        Input in;
        InitEvents(&in);
        SDL_Init(SDL_INIT_VIDEO);               // preapare SDL
        screen = SDL_SetVideoMode(LARGEUR_FENETRE,HAUTEUR_FENETRE,32,SDL_HWSURFACE|SDL_DOUBLEBUF);
        carte = ChargerMap("level2.txt",LARGEUR_FENETRE,HAUTEUR_FENETRE);
        perso = InitialiserSprite(150,150,24,32,carte);
        FocusScrollBox(carte,perso,200,150,400,300);
        while(!in.key[SDLK_ESCAPE])
        {
                UpdateEvents(&in);
                Evolue(&in,perso);
                AfficherMap(carte,screen);
                AfficherSprite(perso,screen);
                SDL_Flip(screen);
                SDL_Delay(5);
        }
        LibererMap(carte);
        SDL_Quit();
        return 0;
}
```

Comme dans l'exemple précédent, j'initialise une carte, puis un perso.
Ensuite, j'appelle la ligne suivante :

```
FocusScrollBox(carte,perso,200,150,400,300);
```

Cette ligne va simplement dire qu'à partir de maintenant, le scrolling sera automatique sur la *carte* autour du sprite *perso* avec un rectangle limite de 200,150,400,300 dans le repère de la fenêtre.

Une fois cette ligne posée, rien à changer de plus dans le main.

Voyons maintenant la structure Map dans le fichier fmap.h

```
struct Smap
{
        int LARGEUR_TILE,HAUTEUR_TILE;
        int nbtilesX,nbtilesY;
        SDL_Surface* tileset;
        TileProp* props;
        tileindex** schema;
        int nbtiles_largeur_monde,nbtiles_hauteur_monde;
        int largeur_fenetre,hauteur_fenetre;
// scroll
        int xscroll,yscroll;
        Sprite* tofocus;
        SDL_Rect rectlimitscroll;
};
```

Les nouveaux paramètres en bas sont un pointeur vers le sprite à focaliser, et le rectangle limite.
Tout ce que va faire la fonction FocusScrollBox, c'est remplir ces paramètres.

```
int FocusScrollBox(Map* m,Sprite* sp,int x,int y,int w,int h)
{
        m->tofocus = sp;
        m->rectlimitscroll.x = x;
        m->rectlimitscroll.y = y;
        m->rectlimitscroll.w = w;
        m->rectlimitscroll.h = h;
        return 0;
}
```

C'est finalement la fonction AfficherMap qui va se charger de mettre le scrolling à jour.

```
int AfficherMap(Map* m,SDL_Surface* screen)
{
        int i,j;
        SDL_Rect Rect_dest;
        int numero_tile;
        int minx,maxx,miny,maxy;
        UpdateScroll(m);
        minx = m->xscroll / m->LARGEUR_TILE-1;
        miny = m->yscroll / m->HAUTEUR_TILE-1;
        maxx = (m->xscroll + m->largeur_fenetre)/m->LARGEUR_TILE;
        maxy = (m->yscroll + m->hauteur_fenetre)/m->HAUTEUR_TILE;
        for(i=minx;i<=maxx;i++)
        {
                for(j=miny;j<=maxy;j++)
                {
                        Rect_dest.x = i*m->LARGEUR_TILE - m->xscroll;
                        Rect_dest.y = j*m->HAUTEUR_TILE - m->yscroll;
                        if (i<0 || i>=m->nbtiles_largeur_monde || j<0 || j>=m->nbtiles_hauteur_mond
e)
                                numero_tile = 0;
                        else
                                numero_tile = m->schema[i][j];<code type="c"><code type="c">
                        SDL_BlitSurface(m->tileset,&(m->props[numero_tile].R),screen,&Rect_dest);
                }
        }
        return 0;
}
```

Vous pouvez constater un appel à une fonction UpdateScroll, qui s'en occupera.

Dans cette fonction :

```
int UpdateScroll(Map* m)
{
	int cxperso,cyperso,xlimmin,xlimmax,ylimmin,ylimmax;
	if (m->tofocus==NULL)
		return -1;
	cxperso = m->tofocus->x + m->tofocus->w/2;
	cyperso = m->tofocus->y + m->tofocus->h/2;
	xlimmin = m->xscroll + m->rectlimitscroll.x;
	ylimmin = m->yscroll + m->rectlimitscroll.y;
	xlimmax = xlimmin + m->rectlimitscroll.w;
	ylimmax = ylimmin + m->rectlimitscroll.h;
	if (cxperso<xlimmin)
		m->xscroll -= (xlimmin-cxperso);
	if (cyperso<ylimmin)
		m->yscroll -= (ylimmin-cyperso);
	if (cxperso>xlimmax)
		m->xscroll += (cxperso-xlimmax);
	if (cyperso>ylimmax)
		m->yscroll += (cyperso-ylimmax);
	ClampScroll(m);
	return 0;
}
```

Nous calculons les nouvelles positions de xscroll et yscroll en fonction du personnage et de la boîte limite, avec les formules vi ci dessus.
Puis nous appelons la fonction ClampScroll

```
void ClampScroll(Map* m)
{
	if (m->xscroll<0)
		m->xscroll=0;
	if (m->yscroll<0)
		m->yscroll=0;
	if (m->xscroll>m->nbtiles_largeur_monde*m->LARGEUR_TILE-m->largeur_fenetre-1)
		m->xscroll=m->nbtiles_largeur_monde*m->LARGEUR_TILE-m->largeur_fenetre-1;
	if (m->yscroll>m->nbtiles_hauteur_monde*m->HAUTEUR_TILE-m->hauteur_fenetre-1)
		m->yscroll=m->nbtiles_hauteur_monde*m->HAUTEUR_TILE-m->hauteur_fenetre-1;
}
```

Cette fonction envisage donc le cas ou la fenêtre de scrolling dépasserait à l'extérieur du monde, et si c'est le cas, elle la remet dans le monde correctement : elle l'empêche de sortir.

Notre scrolling automatique est maintenant en place.
A l'utilisation, il suffira uniquement d'appeler la fonction FocusScrollBox une fois pour toutes et le scrolling sera automatique !
Plus besoin de se casser la tête, de mettre en place plein de lignes dans le main.

Un seul appel de FocusScrollBox , et le tour est joué !

A la fin de cette partie, les collisions avec le décor sont gérées, le scrolling est automatique, et tout cela avec peu d'appels de fonctions.
Si les techniques ont pu vous paraître compliquées, rappelez vous qu'une fois en place, **l'utilisation** est simple.

Après tout, pour le moment, nous n'avons dans notre header fmap.h que 6 fonctions...

Ceci est la troisième version du tutorial sur le TileMapping.

Gardez bien en tête que même si la technique vous paraît complexe, une fois les outils en place, l'utilisation, elle, ne l'est pas trop. Il suffit de regarder la taille du `main` pour le comprendre.

Voici ce qui sera prévu dans les parties suivantes :

- remplacer cet affreux carré vert par un personnage animé ;
- gérer un champ de gravité, pour pouvoir faire des sauts et avancer de façon plus réaliste ;
- faire un fond de stage qui défilera moins vite que les tiles.

Nous pourrons aussi voir :

- comment animer des tiles (eau, lave qui bouge), comment avoir des tiles semi transparents : une barrière par exemple qui laisserait apparaitre le fond entre ses barreaux :
- comment rajouter des ennemis, leur donner un comportement ;
- comment interagir avec le monde (casser des briques, prendre des bonus...) ;
- comment mettre des tiles qui masquent en partie notre personnage : par exemple il pourra passer derrière un bosquet et nous ne verrons alors plus ses jambes.

Nous verrons cela avec le temps, en fonction de mes disponibilité, de vos commentaires, de vos demandes par message privé ou sur le topic officiel du tutoriel (http://www.siteduzero.com/forum-83-471876-p1-tutoriel-tile-mapping.html).

À bientôt pour la suite !
