Programmation de jeu 2D : Un morpion en SDL, Deuxième partie

par Jean Christophe Beyler

Date de publication : 30/03/2006

Dernière mise à jour : 30/03/2006

Dans cette partie, nous allons voir comment ajouter le début du code qui s'occupera du morpion. On verra les concepts de bases pour la gestion de l'affichage et de la souris.

1 - Introduction

Bienvenue à la deuxième partie de ce tutoriel. Nous allons voir comment intégrer au code de la première partie, du code qui permettra d'afficher un plateau de morpion et gérer le clic de la souris.

Nous pourrions faire ce programme en un seul fichier. Et cela montrerait aussi comment faire un morpion, mais si vous essayez de faire des programmes plus compliqués, très rapidement vous allez vous perdre. Nous allons donc présenter comment séparer correctement (enfin, comment je le fais) le code des programmes.

Nous allons prendre le code de la première partie (Voici le lien : .cpp (3 Ko)) et nous allons le compléter pour qu'il affiche correctement le jeu et gère le clic de la souris.

1.1 - Plan

Le nouveau code est réparti en 5 fichiers. Cette page sera décomposée par rapport aux nouveaux fichiers. Voici une présentation rapide :

1 Define.h : Définition des constantes
2 Main.h et Main.cpp : Presque identique à la première version
3 Jeu.h et Jeu.cpp : L'affichage et gestion du plateau de jeu

2 - Le programme

2.1 - Define.h

Tout d'abord, nous allons centraliser les inclusions et les constantes du programme dans un fichier Define.h. Tous les fichiers du projet incluront donc ce fichier s'ils ont besoin des constantes générales du programme.

Centralisation des constantes
```
#ifndef H_DEFINE
#define H_DEFINE

#include <SDL.h>
#include <iostream>

const int WIDTH=600;
const int HEIGHT=600;

#endif
```

2.2 - Les fichiers Jeu

Pour rendre le programme le plus réutilisable possible, nous allons séparer le plus possible le code qui s'occupe du morpion du reste du code. La classe Jeu présentée ici sera donc le point d'entrée du code qui gère la souris et s'occupe de l'affichage du plateau de jeu. L'avantage de faire cette séparation est dans un esprit de portabilité. En effet, tout ce qui n'est pas en rapport avec le morpion peut être réutilisé pour un autre jeu.

En effet, dans la première partie, nous avons montré comment est utilisée la boucle générale SDL. Par exemple, DirectX utilise la même solution mais, bien sûr, les structures et les fonctions ne sont pas les mêmes. En séparant donc notre gestion du jeu de cette boucle, notre programme pourra facilement passer d'un système à l'autre.

Il possède donc certaines fonctions pour afficher/gérer les clics souris ou le clavier de façon indépendante à la représentation ou technique utilisée par SDL/DirectX/Glut/GTK/etc.

2.2.a - Ajout de Jeu.h

Maintenant, à la place d'avoir le code du jeu dans le **main**, nous allons le mettre dans un fichier Jeu.cpp. Pour pouvoir utiliser cette classe dans le fichier Main.cpp, nous avons besoin d'un fichier d'en-tête. Nous commencerons donc par présenter le fichier Jeu.h.

Définition de la classe Jeu
```
#ifndef H_JEU
#define H_JEU

#include "Define.h"

//Enumération des différentes possibilités d'une case du jeu
enum Case {
    Vide=0,
    Rond,
    Croix
};

//Classe du jeu
class Jeu
{
    private:
            //Le plateau jeu
            Case plateau[3][3];
```

Définition de la classe Jeu
```
                //Surfaces d'un rond, d'une croix et d'un fond
                SDL_Surface *o, *x, *bg;

                //Variable pour un tour
                Case tour;

        public:
                //Créateur/Destructeur
                Jeu();
                ~Jeu();

            //Fonction d'initialisation (chargement des surfaces)
            bool init();

                //Gestion du jeu lors d'un clic
                void clic(int , int);

                //Fonction d'affichage
                void aff(SDL_Surface *screen);
    };
    #endif
```

Commençons par expliquer les variables privées de la classe. Nous avons trois groupes de variables pour cette classe :

1 Le plateau de jeu pour le morpion est bien sûr un tableau de dimension [3][3]. Le type d'une case est donné par une énumération **Case**. Chaque case peut être de type *Vide*, *Rond* ou *Croix*;
2 Les surfaces d'un rond, d'une croix et de l'image de fond. Avec la bibliothèque SDL, les images sont définies par des surfaces;
3 Et finalement, pour savoir qui doit jouer, nous utiliserons la variable **tour**. Cette variable pourra donc prendre les valeurs *Rond* ou *Croix*.

2.2.b - Ajout de Jeu.cpp

Présentons, à présent, les fonctions membres de la classe Jeu.

2.2.b.i - Constructeur

Constructeur de Jeu
```
    Jeu::Jeu()
    {
        int i,j;

        //On met toutes les cases à Vide
        for(i=0;i<3;i++)
                for(j=0;j<3;j++)
                    plateau[i][j] = Vide;

        //Valeur par défaut pour les surfaces
        o=NULL;
        x=NULL;
        bg=NULL;
    }
```

En quelques lignes, nous avons initialisé le tableau du jeu a Vide. Ensuite, nous mettons les pointeurs vers les surfaces à NULL. Ce sera le travail de la fonction **init** de charger les images.

2.2.b.ii - Destructeur

Bien sûr, dans le destructeur, nous allons libérer les surfaces allouées (remarquons que, comme pour free/delete,

nous pouvons passer NULL à ces fonctions) :

Destructeur
```
Jeu::~Jeu()
{
    //On libére les surfaces
    SDL_FreeSurface(o);
    SDL_FreeSurface(bg);
    SDL_FreeSurface(x);
}
```

2.2.b.iii - Fonction init

La fonction init permettra de charger les images dans les surfaces de la classe. Si jamais il y a un problème, la fonction rendra **false**.

Fonction init
```
bool Jeu::init()
{
    //Vérification de l'allocation des surfaces
    if(o!=NULL)
        {
            SDL_FreeSurface(o), o = NULL;
        }
    if(x!=NULL)
        {
            SDL_FreeSurface(x), x = NULL;
        }
    if(bg!=NULL)
        {
            SDL_FreeSurface(bg), bg = NULL;
        }

    //On charge toutes les images dans les surfaces associées
    o = SDL_LoadBMP("o.bmp");
    x = SDL_LoadBMP("x.bmp");
    bg = SDL_LoadBMP("bg.bmp");

    //On teste le retour du chargement
    if( (o==NULL) || (x==NULL) || (bg==NULL))
        {
        cout << "Probleme de chargement du O, du X ou de l'image de fond" << endl;
        return false;
        }

    //On initialise le premier tour: ce sera Rond qui commencera
    tour = Rond;

    //Mis en place de la transparence
    if(SDL_SetColorKey(o,SDL_SRCCOLORKEY,0)==-1)
        cout << "Erreur avec la transparence du rond" << endl;

    if(SDL_SetColorKey(x,SDL_SRCCOLORKEY,0)==-1)
        cout << "Erreur avec la transparence de la croix" << endl;

    return true;
}
```

Avant de commencer le chargement des images, nous vérifions d'abord que les images n'ont pas déjà été chargées. Si c'est le cas, nous libérons la mémoire. Je tiens à souligner la technique suivante :

Une bonne habitude
```
    if(o!=NULL)
        {
            SDL_FreeSurface(o), o = NULL;
        }
```

Cela permet d'être sûr que personne ne s'amuse à insérer du code entre la libération et la mise à NULL du pointeur. Ce genre d'erreur est souvent évité en mettant les deux instructions sur la même ligne, separées d'une virgule. C'est d'ailleurs la seule fois où vous me verrez mettre deux instructions sur une même ligne ou utiliser une virgule.

Nous utilisons la bibliothèque SDL pour charger les images du fond d'écran (la grille du morpion), l'image d'un rond et l'image d'une croix. Comme vous le voyez, le chargement d'un fichier BMP se fait simplement. Par contre, il faudra utiliser l'extension de SDL_image pour charger d'autres formats d'image

Remarque, en principe, il est bon d'utiliser la fonction SDL_DisplayFormat pour avoir le même format pour toutes les surfaces. C'est une optimisation facile à mettre en place et importante. Nous ne le faisons pas ici parce que nous voulons d'abord un programme qui fonctionne et ensuite nous procéderons à des optimisations.

Après le chargement, nous vérifions que le chargement s'est bien passé. Ensuite, nous initialisons **tour** à *Rond* pour que le premier joueur joue avec les ronds. Comme beaucoup d'images que nous chargerons pour l'affichage à l'écran, nous avons besoin d'une couleur transparente. Généralement, nous utilisons la couleur magenta (puisqu'elle est rarement utilisée dans les images) mais ici nous utilisons le noir. Donc, pour définir la couleur noire comme couleur transparente, nous utilisons la fonction :

Prototype de SDL_SetColorKey
```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

Cette fonction retourne -1 en cas d'erreur. Comme vous pouvez le remarquer, je ne fais qu'un simple affichage dans le cas d'une erreur. SDL possède un comportement différent dépendant du Système d'Exploitation pour l'affichage texte (via cout sous C++ et via printf/fprintf sous C). Sous Windows, il crée un fichier stdout.txt et stderr.txt et, sous linux, il laisse le comportement natif (donc via le terminal qui a lancé le programme). Pour les problèmes mineurs, j'affiche simplement qu'il y a eu une erreur mais je ne vais pas sortir du programme pour autant.

Revenons à la fonction SDL_SetColorKey, la variable **flag** peut prendre ces masques :

1. SDL_SRCCOLORKEY : pour définir la couleur transparente;
2. SDL_RLEACCEL : pour définir l'accélération RLE si possible. En résumé, cette option permet d'accélérer l'affichage pour des images ayant beaucoup de pixels transparents.

Le dernier paramètre de cette fonction permet de donner la couleur qui deviendra transparente lorsque que nous copierons la surface. Sans rentrer trop dans les détails, il existe trois canaux de couleurs en informatique : le rouge, le vert et le bleu. Mais, nous avons aussi un canal alpha qui servira pour faire du blending (par exemple). Par contre, SDL_SetColorKey demande à ce que ce dernier paramètre soit donné sous la forme d'un entier à 32 bits. Puisqu'il est difficile de calculer directement la correspondance (r,g,b,a) vers ce format à 32 bits, SDL fournit donc une fonction qui permet de le faire :

Prototype de SDL_MapRGB
```
Uint32 SDL_MapRGB(SDL_PixelFormat* fmt, Uint8 r, Uint8 g, Uint8 b)
```

et si nous voulons spécifier le canal alpha :

Prototype de SDL_MapRGBA
```
Uint32 SDL_MapRGBA(SDL_PixelFormat* fmt, Uint8 r, Uint8 g, Uint8 b, Uint8 a)
```

Le premier paramètre est fourni par la surface qui nous intéresse. Donc si nous prenons notre code précédent,

nous avions :

Solution proposée
```
	if(SDL_SetColorKey(x,SDL_SRCCOLORKEY,0)==-1)
		cout << "Erreur avec la transparence de la croix" << endl;
```

Nous aurions pu aussi écrire :

Solution alternative
```
	if(SDL_SetColorKey(x,SDL_SRCCOLORKEY,SDL_MapRGB(x->format,0,0,0))==-1)
		cout << "Erreur avec la transparence de la croix" << endl;
```

2.2.b.iv - Fonction d'affichage

Présentons maintenant la fonction d'affichage. Cette fonction est assez simple, elle affichera l'image de fond, puis parcourt le tableau et affiche chaque case non vide. Puisque nous affichons un jeu de morpion, nous allons diviser la zone d'affichage de la fenêtre en trois colonnes et trois lignes.

Rappelons d'abord son prototype :

Prototype de la fonction d'affichage
```
   void affichage(SDL_Surface *screen);
```

Cette fonction prend donc en paramètre la surface sur laquelle nous voulons dessiner l'image courante. Généralement, nous passerons la surface qui a été rendue par la fonction **SDL_SetVideoMode**.

Pour afficher une surface, nous déclarons un rectangle avec la structure SDL_Rect. En effet, pour afficher une surface on utilise la fonction En fait, ce n'est pas entièrement vrai, SDL_BlitSurface est une définition macro vers une fonction nommée SDL_UpperBlit mais on n'appelle jamais cette dernière directement. .

Prototype de SDL_BlitSurface
```
   int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect);
```

Le premier argument est l'image que l'on veut copier. Le deuxième argument est la partie de la surface source que nous voulons copier (si nous mettons NULL, toute la surface source est utilisée). Le troisième argument est la surface destination et le quatrième argument est la position que va prendre la copie de src (s'il est égal à NULL, alors la position est (0,0)).

Je dis bien "position". En effet, la version actuelle de SDL n'utilise pas la taille du rectangle pour faire l'affichage. Il n'y a donc pas de zoom possible avec la bibliothèque SDL de base. Il faudra utiliser l'extension SDL_gfx pour faire des zooms/rotations ou alors faire les transformations à la main.

Donc la fonction d'affichage commence par le calcul de la largeur et hauteur d'une case et l'initialisation de la structure **r** qui sera la position où copier les ronds et les croix.

La fonction d'affichage
```
   void Jeu::aff(SDL_Surface *screen)
   {
       //Le couple (w,h) représentera les dimensions d'une case du plateau
       int w = WIDTH/3, h=HEIGHT/3,i,j,k,l;
       SDL_Rect r = { 0 };

       //Dessiner le fond d'ecran
       SDL_BlitSurface(bg,NULL,screen,&r);
```

La fonction d'affichage

```
    //On parcourt les cases du tableau, r sera le SDL_Rect qui représentera la position de la case
courante
    for(k=0,i=0;i<HEIGHT;i+=h,k++)
            {
            r.y = i;
            for(j=0,l=0;j<WIDTH;j+=w,l++)
                {
                r.x = j;

                //On dessine en fonction du type de la case
                if(plateau[k][l]==Croix)
                    {
                    SDL_BlitSurface(x, NULL, screen,&r);
                    }
                else if(plateau[k][l]==Rond)
                {
                    SDL_BlitSurface(o, NULL, screen,&r);
                    }
            }
        }
}
```

Dans cette fonction d'affichage, nous commençons par la copie de l'image de fond. En principe, on met d'abord une couleur de fond (généralement le noir) sur toute la surface de la fenêtre, mais ici, notre image de fond occupera toute la fenêtre, donc cette mise à zéro est inutile.

Ensuite, nous avons un nid de boucles qui permet de parcourir le plateau du jeu. Nous allons donc pouvoir dessiner les ronds et les croix. Nous allons avoir deux variables par dimensions : le couple (k,l) parcourrait le tableau jeu de la classe et le couple (i,j) représentera la position courante de la case à afficher.

Puis, le corps de la boucle met à jour la position de la variable **r** (la position destination de la surface copiée). Finalement, un simple test vérifie si la case courante est une croix ou un rond et affiche la surface associée.

Pourquoi avoir deux couples ? Pour une petite raison d'optimisation et simplification du code. Nous pouvons facilement remarquer, qu'à tout moment de cette boucle, nous avons l'équivalence :

Rapport entre (i,j) et (k,l)

```
    i == k*h
    j == l*w
```

Donc nous pourrions écrire la mise à jour de la position de r comme ceci :

Solution alternative

```
    r.x = l*w;
    r.y = k*h;
```

Mais, nous échangeons donc une paire de sommes par deux multiplications. Une multiplication prenant plus de temps qu'une somme, je préfère limiter leur nombre. Par contre, il est déconseillé de trop tenter d'optimiser lors d'un début de projet. Les déclarations de boucles **for** avec plusieurs itérateurs sont généralement des mauvaises idées. Je me le permets ici parce ce que cela ne complique pas vraiment le code ou sa lecture.

2.2.b.v - Fonction clic

La fonction clic gère l'évènement souris. Rappelons d'abord son prototype :

Prototype de la fonction clic

```
void clic(int x, int y);
```

Elle prend comme argument les coordonnées du clic souris. Comme pour l'affichage, nous allons commencer par calculer la largeur et hauteur d'une case.

Calcul de la dimension d'une case
```
	//On récupère la largeur et l'hauteur d'une case
	w = WIDTH/3;
	h = HEIGHT/3;
```

Ensuite, nous calculons la case associée à la position de la souris lors du clic avec le calcul suivant :

Transformation (x,y) -> (i,j)
```
	//Calcul de la case associée
	i = y/h;
	j = x/w;
```

Chaque case du morpion fait w pixels de large et h pixels de haut. Logiquement, si on prend la division entière (y/h, x/w) nous aurons la case associée. Remarquez l'inversion des coordonnées. Ceci arrive souvent lorsque nous passons de l'affichage à l'interprétation des données du jeu. En effet, par convention en C/C++ et d'autres langages, la 1ère dimension d'un tableau à 2 dimensions est considérée comme les lignes du tableau. Donc (2, 3) représente la case qui se trouve à la 3ème ligne (on commence à zéro) et la 4ème colonne. Or, à l'affichage, (53, 123) représente souvent le 54ème pixel (on commence aussi à zéro!) de la 124ème ligne! Il faut donc faire attention et programmer en connaissance de cause.

Enfin, nous allons mettre à jour la valeur de la case **si** la case est de type *Vide*. Ahhh, après tant de discussions, nous arrivons à la première ligne de code qui est intimement liée au fait que nous programmons un morpion. En effet, le fait que le clic n'est pris en compte **que** si la case est vide est déjà un bon point. Remarquons qu'après la mise à jour de la case (si elle est effectuée), nous mettons également à jour la variable **tour**.

Mise a jour du plateau de jeu
```
	//Si la case est vide, on met à jour son type et la variable tour
	if(plateau[i][j]==Vide)
	    {
	        plateau[i][j] = tour;
	    tour = (tour==Rond)?Croix:Rond;
	    }
```

2.3 - Les fichiers Main

Nous arrivons au fichier Main.cpp. D'abord, nous allons ajouter un fichier Main.h. J'ai l'habitude d'avoir une seule inclusion dans les fichiers sources vers un fichier d'en-tête du même nom. Donc, bien que le fichier Main.cpp ne nécessite que l'inclusion de Define.h, je crée quand même le fichier Main.h (c'est aussi pour simplifier mon makefile...).

2.3.a - Ajout de Main.h

Le fichier Main.h
```
#ifndef H_MAIN
#define H_MAIN

#include "Define.h"

#endif
```

2.3.b - Modification de Main.cpp

Le Main.cpp commence par l'inclusion des fichiers d'entête Main.h et Jeu.h. Ensuite, nous déclarons la variable globale **jeu**. Personnellement, je ne suis pas contre l'utilisation des variables globales tant que nous les réduisons au minimum.

Début du fichier Main.cpp
```
#include "Main.h"
#include "Jeu.h"
Jeu jeu;
```

Ensuite, l'initialisation SDL ne change pas par rapport à la première version. On ajoute tout de même un appel vers la fonction **init** de l'instance **jeu**. Si jamais l'initialisation de la fonction **init** retourne false, nous sortons de la fonction **main**, mettons une fin au programme.

Initialisation du jeu
```
        if(!jeu.init())
            return 1;
```

Puis, dans la boucle générale, deux lignes sont ajoutées à la fin de la boucle :

Ajout de l'appel d'affichage et SDL_Flip
```
        jeu.aff(screen);
        SDL_Flip(screen);
```

Nous demandons au jeu de dessiner l'état du morpion à l'écran. Une fois que le l'instance **jeu** dessine le plateau de jeu, nous appelons la fonction SDL_Flip. Je vais maintenant prendre un peu plus de détails sur le double buffering.

Le double buffering est une technique pour améliorer l'affichage d'un jeu. Ce qu'il faut savoir, c'est que la carte video affiche l'information contenue dans la mémoire video et, qu'en même temps, à travers l'appel jeu.aff(screen), nous calculons les couleurs des pixels de la fenêtre.

Forcément, si nous modifions des pixels pendant que la carte video les affiche, il peut y avoir un problème de synchronisation, ce qui peut se traduire par un scintillement. La solution est donc d'afficher une image pendant qu'on travaille sur une autre image. Lorsque le calcul de la prochaine image est finie, on "flip" (retourne en français) les images.

L'utilisation du mot flip se traduit par l'analogie qu'on vous montre le verso d'une feuille pendant que le programme dessine sur le recto. Lorsque le dessin est fini, on retourne la feuille et on recommence à travailler sur le recto pendant que vous regardez le "nouveau" verso.

Le dernier changement dans la fonction main se trouve dans la boucle événementielle. Nous y ajoutons la gestion du clic souris. Lorsqu'un tel évènement se produit, nous appelons la fonction membre clic de la classe Jeu en lui transmettant les coordonnées de la souris.

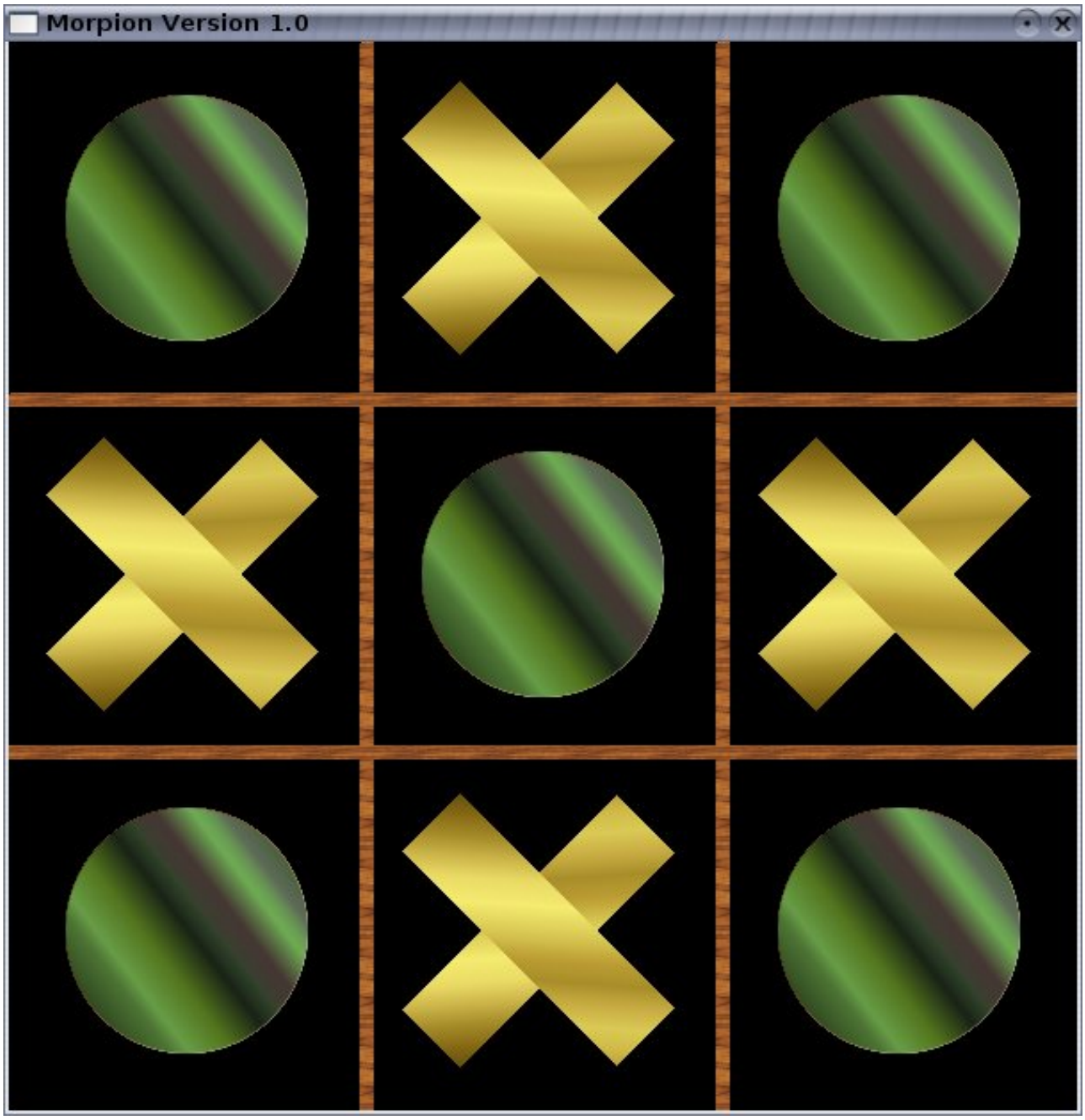
Ajout de l'événement clic de souris
```
        case SDL_MOUSEBUTTONUP:
                jeu.clic(event.button.x, event.button.y);
            break;
```

3 - Fichiers de données

L'utilisation de fichiers bitmap pour l'affichage ajoute donc à ce projet des fichiers de données. Nous verrons par la suite d'autres exemples de fichiers de données mais sachez que leur utilisation permet une grande souplesse dans le projet. En effet, il est possible de modifier les fichiers de données pour modifier le comportement du programme de base. Tout cela, sans devoir recompiler le programme!

C'est une technique souvent utilisée. Dans notre cas, cela nous permet de changer à volonté les fichiers *o.bmp*, *x.bmp* et *bg.bmp* à volonté sans devoir tout recompiler.

Et enfin, le moment tant espérer, voilà la première image du morpion en action. C'est une image provisoire car il y aura encore beaucoup de changements (Les images de la grille, des ronds et des croix ont été créées par Gimp)!



4 - Conclusion

Ceci termine donc cette deuxième partie de l'élaboration du morpion. Nous avons maintenant un programme qui affiche l'état du jeu et gère les clics. En effet, lorsque nous cliquons sur une case vide, nous mettons la case à jour et l'affichage se fait en conséquence. Par contre, si la case est déjà occupée, le clic est ignoré. Dans la prochaine partie, nous verrons comment ajouter les règles du morpion. Cela permettra de vérifier si le jeu est terminé, de décider qui a gagné et de recommencer la partie.

Jc

- Sommaire du tutoriel;
- Ouvrir une fenêtre SDL;
- Lier la souris à la fenêtre et afficher des ronds à l'endroit cliqué;
- Ajouter les règles du jeu et le finaliser;
- Ajouter les classes Objet et Moteur pour rendre le jeu plus souple;
- Ajouter un menu dans un jeu;
- Ajouter une Intelligence Artificielle (Min-Max);
- Ajouter une Intelligence Artificielle (Alpha-Beta);

5 - Téléchargements

Voici le code source de ce tutoriel : zip (108 Ko)

Voici la version pdf de cet article : (120 Ko).