

Programmation de jeu 2D: Le morpion, Troisième partie

par [Jean Christophe Beyler](#)

Date de publication : 04/04/2006

Dernière mise à jour : 04/04/2006

Dans cette partie du tutoriel, nous allons voir comment ajouter les règles du morpion et comment recommencer une partie sans devoir relancer le programme. A la fin de cette partie, nous aurons une première version d'un morpion qui fonctionne correctement et nous prévient si quelqu'un a gagné.

- 1 - Introduction
 - 1.1 - Présentation de cette partie
 - 1.2 - Qu'est-ce que la logique de jeu?
- 2 - Le programme
 - 2.1 - Modification du fichier Jeu.h
 - 2.1.a - Changement de l'énumération Case
 - 2.1.b - Modification des membres de la classe Jeu
 - 2.1.c - Ajout de fonctions membres dans la classe Jeu
 - 2.1.c.i - La fonction videJeu
 - 2.1.c.ii - La fonction verifFin
 - 2.1.d - Modification des fonctions membres de Jeu
 - 2.1.d.i - Le constructeur et destructeur
 - 2.1.d.ii - La fonction clic
 - 2.1.d.iii - La fonction aff
- 3 - Conclusion
- 4 - Téléchargements

1 - Introduction

Bienvenue dans la troisième partie du tutoriel sur le jeu du morpion. Nous allons voir comment ajouter un peu de logique de jeu et comment intégrer une remise à zéro d'un jeu (sans devoir relancer le programme!).

A la fin de la deuxième partie de ce tutoriel, nous avons un programme qui affichait un quadrillage de morpion (image de fond). Lorsque nous cliquons dans la fenêtre, le programme est capable d'afficher les ronds et les croix des deux joueurs.

Voyons maintenant comment ajouter un peu de logique de jeu et rendre ce morpion un peu plus correct.

1.1 - Présentation de cette partie

Dans cette troisième partie, nous allons ajouter les règles associées au morpion. Il faudra donc pouvoir vérifier si le jeu est fini et pouvoir déterminer qui a gagné (ou s'il y a un match nul).

Nous allons compléter le code du deuxième tutoriel. Voici le lien vers ce code: [zip \(105 Ko\)](#).

1.2 - Qu'est-ce que la logique de jeu?

J'appelle "logique de jeu" toutes les règles du jeu. Par exemple, pour un morpion, si trois ronds forment une ligne alors les ronds gagnent. Ceci fait partie de la logique du jeu. Donc dans la logique du jeu, nous devons pouvoir savoir si le jeu est fini, qui a gagné (ou déterminer qu'il y a un match nul).

Graphiquement, nous traduirons cela par l'affichage d'une autre surface pour montrer à l'utilisateur que quelqu'un a gagné.

2 - Le programme

Aucun fichier ne sera ajouté par rapport à la deuxième version du programme. De plus, le fait d'avoir séparé correctement le lancement du programme et la boucle générale de la gestion du morpion, aucun changement ne sera apporté aux fichiers Main.cpp, Main.h et Define.h. Regardons donc les changements des fichiers Jeu.h et Jeu.cpp.

2.1 - Modification du fichier Jeu.h

Nous allons modifier l'énumération **Case** pour ajouter un type *Gagne* et ajouter quelques fonctions dans la classe Jeu pour gérer les règles du morpion. Ce qui suit est une présentation plus détaillée.

2.1.a - Changement de l'énumération Case

Dans le fichier Jeu.h, nous avons ajouté un élément dans l'énumération des types de cases pour avoir le type *Gagne*. Ce nouveau type servira pour pouvoir afficher une surface différente pour les cases qui forment une ligne.

Ajout de l'énumération Gagne

```
enum Case {
    Vide=0,
    Rond,
    Croix,
    Gagne
};
```

2.1.b - Modification des membres de la classe Jeu

Ensuite, nous ajoutons deux surfaces pour pouvoir dessiner les cases *Gagne* pour les ronds et les croix. Nous ajoutons aussi une variable *typegagne* de type *Case* pour nous donner la personne qui a gagné (s'il y en a une). Finalement, nous avons un booléen *fini* qui nous dit si le jeu est fini.

Membres de la classe Jeu

```
//Surfaces d'un rond, d'une croix, d'un fond et les surfaces gagnées
SDL_Surface *o, *x, *bg, *gagneo, *gagnex;

//Variable pour un tour et pour savoir qui a gagné
Case tour, typegagne;

//Booléen pour savoir si la partie est terminée
bool fini;
```

N'est-ce pas redondant d'avoir un booléen **fini** et une variable **typegagne**?

On pourrait par exemple dire: Si *typegagne != Vide* alors quelqu'un a gagné donc le jeu est fini. Mais alors comment définir le match nul?

On pourrait répondre que si *typegagne == Gagne*, alors c'est une égalité...

Je n'aime pas ce genre de chose. Il est bon de séparer (tant qu'on a le luxe d'avoir assez de mémoire) les choses correctement. Nous avons donc un booléen qui nous informe si le jeu est fini et une autre variable pour savoir qui a gagné (ou match nul).

Alors la convention prise pour ce jeu est de dire:

- 1 Rond a gagné si (**fini==true**) && (**typegagne==Rond**);
- 2 Croix a gagné si (**fini==true**) && (**typegagne==Croix**);
- 3 Egalité si (**fini==true**) && (**typegagne==Vide**);
- 4 Pas encore fini si (**fini==false**).

2.1.c - Ajout de fonctions membres dans la classe Jeu

Ensuite nous ajoutons deux fonctions à la classe:

Ajout de fonctions de la classe Jeu

```
//Fonction pour vider le plateau de jeu
void videJeu();

//Fonction pour vérifier si la partie est terminée
void verifFini();
```

2.1.c.i - La fonction videJeu

La fonction **videJeu** remet toutes les valeurs à zéro. Elle sera appelée dans le constructeur pour éviter une redondance de code. En fait, elle n'a rien de nouveau, ce n'est qu'une migration de code qui se trouvait dans le constructeur (on y ajoute tout de même la mise à false de la variable **fini**). Cette fonction est plus importante que ce qu'on pourrait croire. Elle permettra de recommencer le jeu sans quitter le programme. Plus le jeu est compliqué, plus cette fonction sera difficile à écrire mais sera toujours aussi importante.

Fonction videJeu

```
void Jeu::videJeu()
{
    int i,j;
    //On vide le plateau
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            plateau[i][j] = Vide;

    //On commence par rond
    tour = Rond;

    //On met fini à false
    fini = false;
}
```

2.1.c.ii - La fonction verifFini

La deuxième fonction la plus importante pour un jeu est une fonction de type **verifFini** qui permet de savoir si le jeu est terminé. De plus, si la fonction est capable de savoir qui a gagné, alors c'est la cerise sur le gâteau. Nous n'allons pas montrer tout le code de cette fonction, seulement le début puisque c'est toujours la même chose:

Nous vérifions les lignes, les colonnes puis les diagonales. Si la première case est non vide, alors nous regardons le reste de la ligne/colonne/diagonale. S'ils sont tous du même type, alors nous savons que quelqu'un a gagné. Nous mettons donc les cases en question à **Gagne** et nous mettons **typegagne** et **fini** à jour.

Test d'alignement par ligne dans verifFini

```
for(i=0;i<3;i++)
{
    if( plateau[i][0]!=Vide )
    {
        //Tester la ligne
        if((plateau[i][0]==plateau[i][1]) && (plateau[i][0]==plateau[i][2]))
        {
```

Test d'alignement par ligne dans verifFini

```

        fini = true;
        typegagne = plateau[i][0];
        plateau[i][0] = Gagne;
        plateau[i][1] = Gagne;
        plateau[i][2] = Gagne;
    }
else
    {
        casevide = true;
    }
}

```

Rappelons que le jeu se termine aussi s'il n'y a plus de cases vides. Dans cette fonction, nous avons donc un booléen **casevide** qui est initialisé à faux et qu'on met à vrai dès qu'on voit une case vide. Vu qu'on teste que la première case des lignes, des colonnes et des diagonales pour les cases vides, nous devons aussi regarder les cases qui restent. Ceci est fait à la fin de la fonction, avant de mettre à jour la variable **fini**.

Fin du test pour le booléen casevide dans verifFini

```

//Dernières cases
if( (plateau[1][1] == Vide)|| (plateau[1][2] == Vide)||
    (plateau[2][1] == Vide)|| (plateau[2][2] == Vide))
    casevide = true;

fini = !casevide || fini;

```

Le test booléen dit bien: le jeu est terminé si aucune case n'est vide ou si fini a déjà été mis à vrai (cela veut dire qu'on a eu un alignement).

2.1.d - Modification des fonctions membres de Jeu

Finalement, le fait d'ajouter ces informations dans le programme a affecté toutes les fonctions de la classe Jeu. Nous avons de la chance puisque cette classe est encore relativement petite. Regardons ensemble les différences apportées.

2.1.d.i - Le constructeur et destructeur

Les seules deux modifications du constructeur de la class **Jeu** est la mise à NULL des variables **gagnex** et **gagneo** et la migration du code vers la fonction **videJeu**.

Modification du constructeur de la classe Jeu

```

Jeu::Jeu()
{
    //Vider le Jeu
    videJeu();

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

```

Du côté destructeur, nous ajoutons simplement la désallocation des nouvelles surfaces.

Modification du destructeur de la classe Jeu

```

Jeu::~Jeu()
{

```

Modification du destructeur de la classe Jeu

```
//On libère les surfaces
SDL_FreeSurface(o);
SDL_FreeSurface(bg);
SDL_FreeSurface(x);
SDL_FreeSurface(gagneo);
SDL_FreeSurface(gagnex);
}
```

2.1.d.ii - La fonction clic

Très peu de changements sont effectués à la fonction **clic**. On ajoute un test dans le cas où **fini** est mis à vrai. La décision pour le moment est de simplement remettre le jeu à zéro.

Remise à zéro

```
if(fini)
{
    videJeu();
}
```

Sinon, on gère le clic de la même façon que dans le dernier tutoriel mais on ajoute un appel à la fonction **verifFini**. Donc, à chaque clic, on vérifie si la partie est terminée.

2.1.d.iii - La fonction aff

On ajoute dans cette fonction la gestion des cases de type "Gagne". Ceci se fait en ajoutant deux if-else ou alors un switch:

Gestion de l'affichage des cases du morpion

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

On aurait pu ajouter deux éléments dans l'énumération de la Case: GagneX et GagneO à la place d'un type Gagne. C'est une solution acceptable, et vous pouvez, si vous voulez, tenter de le faire comme exercice!

3 - Conclusion

En conclusion, on remarque que finalement les changements ne sont pas extraordinaires pour ajouter les règles du jeu du morpion et les intégrer dans le programme. En effet, avec quelques ajustements, on peut les gérer de façon intéressante et claire. Dans la prochaine partie, nous montrerons comment abstraire le moteur du jeu par rapport à tout ce qui se trouve en-dessous, afin de bien séparer les différentes parties du programme.

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 \(Alpha-Beta\)](#);

4 - Téléchargements

Voici le code source de ce tutoriel: [zip \(127 Ko\)](#)

Voici la version pdf: [pdf \(40 Ko\)](#)