

# Programmation de Jeux 2D: Un pong en SDL/OpenGL, Deuxième partie

par [Jean Christophe Beyler](#)

Date de publication : 25/09/2006

Dernière mise à jour : 25/09/2006

Cette partie pose les fondations du jeu du pong. A travers 3 classes, nous allons pouvoir définir les interactions nécessaires pour la gestion d'un jeu comme le pong. Une classe Objet pour chaque balle (et bientôt chaque raquette), une classe pour le jeu et une classe Moteur pour diriger le tout. Cet article présente donc ces classes.

- 1 - Introduction
- 2 - La modification du fichier Main.cpp
  - 2.1 - La classe Moteur
  - 2.2 - Gérer la fréquence d'affichage
    - 2.2.a - Version basique
    - 2.2.b - La version SDL\_gfx
- 3 - La classe Moteur
  - 3.1 - Moteur.h
  - 3.2 - Moteur.cpp
- 4 - La classe Jeu
  - 4.1 - Le fichier Jeu.h
    - 4.1.a - Les champs membres
    - 4.1.b - Les fonctions membres
  - 4.2 - Le fichier Jeu.cpp
    - 4.2.a - Le Constructeur et le Destructeur
    - 4.2.b - La fonction initTextures
    - 4.2.c - Les autres fonctions d'initialisation
    - 4.2.d - La fonction addObjet
    - 4.2.e - Les fonctions d'affichage
    - 4.2.f - La fonction updateObjets
    - 4.2.g - La fonction gereSceneServeur
- 5 - La classe Objet
  - 5.1 - Le fichier Objet.h
    - 5.1.a - Les champs membres
    - 5.1.b - Le fichier Objet.cpp
- 6 - La classe Vecteur
- 7 - La classe Rand
- 8 - Conclusion
- 9 - Téléchargements
- 10 - Remerciements

## 1 - Introduction

Bienvenue à cette deuxième partie sur la programmation du pong. Nous allons de nouveau ajouter beaucoup de choses dans cette partie mais rien de très surprenant. Si vous avez suivi la série du [morpion](#), vous avez vu comment faire une classe **Objet**, une classe **Jeu** et une classe **Moteur**.

Cette partie présentera donc ces trois classes et leur interaction dans ce programme. Il y a beaucoup de choses à montrer donc je m'excuse déjà pour la longueur de cet article.

## 2 - La modification du fichier Main.cpp

Il n'y a pas beaucoup de changements dans ce fichier. Mais nous allons présenter les quelques endroits où il y a des différences dans cette section.

### 2.1 - La classe Moteur

Dans ce fichier Main.cpp, nous déclarons une variable globale :

#### Déclaration d'une variable globale

```
//Variable globale
Moteur moteur;
```

Comme vous le voyez, le programme aura une instance de la classe Moteur. Cette classe permettra de faire le lien entre le jeu et la fonction **main**. Nous verrons la déclaration de cette classe plus loin dans cet article.

Nous avons un appel à une fonction **init** de la classe moteur. Cet appel permettra de gérer toute l'initialisation du moteur.

#### Initialisation de la classe moteur

```
//Initialisation du moteur
if(moteur.init()==false)
    done = 1;
```

## 2.2 - Gérer la fréquence d'affichage

### 2.2.a - Version basique

La bibliothèque SDL ne contient pas directement des fonctions pour gérer la fréquence d'affichage. En suivant la réponse de la question FAQ [Comment gérer la vitesse d'affichage ?](#), nous allons mettre le code qui permet de limiter la fréquence d'affichage. Nous commencerons donc par initialiser la variable **checkTime** avant la boucle événementielle.

#### Initialisation de la variable checkTime

```
checkTime = SDL_GetTicks();
```

Ensuite, nous allons entourer le code d'affichage et le code de gestion de la variable **fps** avec le code de la FAQ :

#### Gestion de l'affichage

```
if(SDL_GetTicks() > (checkTime + 1000 / wantedfps) )
{
    // On met a jour la variable checkTime
    checkTime = SDL_GetTicks();

    // On incremente la variable fps
    fps++;
    // Gerer l'affichage du fps
    now = time(NULL);
    if(now!=last)
    {
        cout << "FPS: " << fps/(now-last) << endl;
        last = now;
        fps = 0;
    }
}
```

### Gestion de l'affichage

```
// Demander au moteur de dessiner la scene
moteur.gereScene();
}
else
{
// Attendre 5 millisecondes
SDL_Delay(5);
}
```

Comme vous le voyez, on ira dans le corps du if tous les **1000/wantedfps** millisecondes. C'est maintenant à travers de la fonction membre **gereScene** que l'affichage et la gestion du jeu se fera.

Vous remarquerez que nous ajoutons une clause **else**. Ce code permet d'arrêter l'exécution pendant 5 millisecondes pour donner un peu de temps au système d'exploitation.

### 2.2.b - La version SDL\_gfx

Si vous voulez une technique un plus facile à gérer, il vous faudra utiliser la bibliothèque SDL\_gfx. En effet, cette extension permet d'avoir des fonctions en plus, dont celles qui vont nous servir ici. Les fonctions qui doivent être utilisées pour gérer la fréquence d'affichage sont définies dans le fichier d'en-tête **SDL\_framerate.h** (par contre, utiliser l'option **-ISDL\_gfx** pour la compilation).

Voici les fonctions définies dans ce fichier :

#### Les fonctions définies dans le fichier SDL\_framerate.h

```
void SDL_initFramerate(FPSmanager * manager);
int  SDL_setFramerate(FPSmanager * manager, int rate);
int  SDL_getFramerate(FPSmanager * manager);
void SDL_framerateDelay(FPSmanager * manager);
```

Toutes ces fonctions utilisent un paramètre de type FPSmanager. Pour chaque programme, vous en définissez un (allouer statiquement ou dynamiquement) que vous passerez à chaque fonction. On va montrer maintenant comment on les utilise.

On commencera par définir une variable de type FPSmanager. Ensuite, en appelant **SDL\_initFramerate**, on initialise la structure.

#### Définition et initialisation de FPSmanager

```
FPSmanager manager;

//Initialisation
SDL_initFramerate(&manager);
```

Par défaut, l'utilisation des ces fonctions mettent en place 30 images par seconde. Pour définir un autre taux d'affichage, il faut utiliser la fonction **SDL\_setFramerate**. Pour récupérer le taux d'affichage, on utilise **SDL\_getFramerate**.

#### Mettre 60 images par seconde

```
//Mettre le nombre d'images par secondes souhaite
SDL_setFramerate(&manager, 35);
```

Bien sûr, si le programme n'arrive pas à garder un taux d'affichage aussi élevé, cette technique (comme la version naïve) ne pourra pas faire mieux. Il faut donc s'assurer que le code de rendu est assez rapide.

Enfin, pour mettre en place la gestion du taux d'affichage, on fait un appel à **SDL\_framerateDelay**. Vous devez faire un appel pour chaque itération de la boucle globale. Cela ressemblerait donc à :

#### Mettre 60 images par seconde

```
while(!done)
{
    ...
    // Demander au moteur de dessiner la scene
    moteur.gereScene();
    SDL_framerateDelay(&manager);
}
```

## 3 - La classe Moteur

Nous allons donc présenter la classe **Moteur**. Ceci sera fait en deux parties, d'abord le fichier d'en-tête puis le fichier source.

### 3.1 - Moteur.h

#### Déclaration de la classe

```
class Moteur
{
private:
    // Le jeu
    Jeu* jeu;

    // Sommes-nous dans le menu ?
    bool dans_menu;

    // Dessiner le jeu
    void dessineJeu();
    // Dessiner le menu
    void dessineMenu();

public:
    // Le constructeur et destructeur
    Moteur();
    ~Moteur();

    // Gerer la scene (affichage + mise a jour)
    void gereScene();

    // Fonction d'initialisation
    bool init();
};
```

Comme vous le voyez, dans cette version du moteur du jeu, nous avons deux variables.

- Nous avons une instance **jeu** de la classe **Jeu**. Comme vous le pouvez le voir, cette classe **Moteur** ne possède rien de plus sur le jeu sous-jacent. Ceci est logique. Nous voulons garder cette classe aussi indépendante que possible.
- Nous avons aussi un booléen **menu** qui permettra de savoir si nous sommes dans le menu ou dans le jeu. Bien que dans cette version, le menu ne sera pas implémenté, je pensais que ce serait bien de le mettre tout de suite en place.

Cette classe possède aussi 4 fonctions membres (dont 2 sont en **private**):

- **gereScene** : cette fonction est le point d'entrée pour l'affichage. Dépendant de la valeur du booléen **menu** on pourra appeler **dessineJeu** ou **dessineMenu**;
- **dessineJeu** : cette fonction se charge de l'affichage lorsque le programme est en mode jeu;
- **dessineMenu** : cette fonction se charge de l'affichage lorsque le programme est en mode menu;
- **init** : cette fonction gère l'initialisation de cette classe et de la classe **init**.

### 3.2 - Moteur.cpp

Cette classe n'est qu'à ses débuts, elle est assez vide pour le moment! Mais nous allons tout de même présenter les fonctions de cette classe.

#### Constructeur et Destructeur

```
//Constructeur
Moteur::Moteur()
```

### Constructeur et Destructeur

```

{
    menu = false;
    jeu = new Jeu();
}

//Destructeur
Moteur::~Moteur()
{
    delete jeu;
}
    
```

Comme vous le voyez, la seule initialisation est de mettre la variable **menu** à **false** et allouer de la mémoire pour l'instance **jeu**. Le programme ira donc directement dans le mode jeu comme nous pouvons le voir dans la fonction **gereScene** :

### La fonction gereScene

```

void Moteur::gereScene()
{
    // Effacer le tampon des couleurs
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();

    if(!menu)
        dessineJeu();
    else
        dessineMenu();

    SDL_GL_SwapBuffers();
}
    
```

Ensuite, viennent le tour des fonctions **dessineMenu** et **dessineJeu**. Pour le moment, la fonction **dessineMenu** est vide. Par contre, le code de la fonction **dessineJeu** contient les appels pour la classe jeu.

### Les fonctions dessineMenu et dessineJeu

```

void Moteur::dessineMenu()
{
}

void Moteur::dessineJeu()
{
    jeu->affiche();
    jeu->gereSceneServeur();
}
    
```

**A quoi sert la fonction gereScene ?** Pour pouvoir ajouter des balles, mettre à jour les positions des balles, vérifier les collisions, ...

**Pourquoi ne pas mettre ce code dans la fonction d'affichage ?** C'est en effet une chose souvent faite. A chaque affichage, on met à jour la position des objets. Mais ce n'est pas très portable. Par exemple, un client du jeu a juste besoin d'avoir les positions des objets et les afficher. Le client n'a pas besoin de mettre à jour les objets, c'est le travail du serveur (bien sûr, dans certaines applications le client fait aussi une partie du travail).

Finalement, nous avons la fonction d'initialisation du moteur :

### La fonction init

```

bool Moteur::init()
{
    return jeu->init();
}
    
```

Cette fonction appelle simplement la fonction d'initialisation pour l'instance du jeu et retourne son résultat.



## 4 - La classe Jeu

La classe **Jeu** possède toutes les informations du jeu. Nous allons présenter la classe **Jeu** dans cette section.

### 4.1 - Le fichier Jeu.h

#### 4.1.a - Les champs membres

Nous allons commencer par la présentation des champs membres de la classe Jeu :

```
//Les objets
std::vector<Objet> objets;

//Pour la creation des balles
unsigned int last;
```

Comme vous le voyez, nous avons un vecteur **objets** pour les objets définis par une constante (qui se trouvera dans le fichier **Define.h**). Le type **vector** nous permet bien sûr de pouvoir facilement ajouter des objets dans le jeu et aussi d'en enlever sans difficulté.

On utilise la variable **last** pour savoir quand ajouter une nouvelle balle dans la partie. Nous verrons plus en détail son utilité lors de la prochaine section.

#### 4.1.b - Les fonctions membres

Voici les fonctions membres de la classe **Jeu** :

##### Les fonctions membres

```
private:
    ...

    //Initialisation des objets et des textures
    bool initObjets();
    bool initTextures();

public:
    //Constructeur et Destructeur
    Jeu();
    ~Jeu();

    //Initialisation de la classe
    bool init();

    //Ajouter un objet dans le jeu, retourne l'indice de l'objet ajoute
    int addObjet(Objet &o);
    //Retourner le nombre d'objets dans le jeu
    unsigned int getNObjets();

    //Afficher le jeu
    void affiche();
    void affObjets();

    //Mettre a jour les objets
    void updateObjets();

    //Gere la scene (mise a jour)
    void gereSceneServeur();
```

Nous avons mis les deux fonctions **initObjets** et **initTextures** en privé puisque ce sont des fonctions qui devraient être uniquement appelées par la fonction **init**

Je ne pense pas que ce soit nécessaire d'en dire plus sur ces fonctions, les commentaires doivent suffire.

## 4.2 - Le fichier Jeu.cpp

Montrons l'implémentation de la classe **Jeu**. A part la fonction **initTextures**, il n'y a pas grand chose de difficile dans cette classe mais je pense que c'est important de présenter les grands points.

### 4.2.a - Le Constructeur et le Destructeur

#### Constructeur et Destructeur

```
//Constructeur
Jeu::Jeu()
{
    last = 0;
}

//Destructeur
Jeu::~Jeu()
{
}
```

Comme vous le voyez le constructeur met simplement à zéro les variables **last**.

### 4.2.b - La fonction initTextures

La fonction **initTextures** sert à initialiser les textures. Elle retourne **true** si tout s'est bien passé et **false** s'il y a eu un problème. Voici le code de cette fonction :

#### La fonction initTextures

```
bool Jeu::initTextures()
{
    int i,j,cnt,pent;
    GLuint txtballe;

    unsigned char *pixels, *tmppixels;

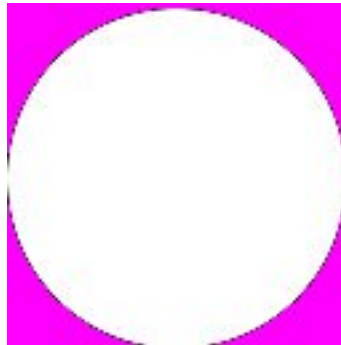
    //On charge l'image "data/balle.bmp"
    SDL_Surface *tmpballe = SDL_LoadBMP("data/balle.bmp");
    if(tmpballe==NULL)
    {
        std::cerr << "Erreur dans le chargement de l'image de la balle" << std::endl;
        return false;
    }

    //On cree une surface RGBA de meme dimension
    SDL_Surface *balle = SDL_CreateRGBSurface(SDL_SWSURFACE, tmpballe->w, tmpballe->h, 32,
        0x000000ff, 0x0000ff00, 0x00ff0000, 0xff000000);

    if(balle==NULL)
    {
        std::cerr << "Erreur dans la creation de l'image de la balle" << std::endl;
        return false;
    }
}
```

Pour dessiner les balles du jeu, nous allons utiliser une texture OpenGL. Ces textures sont définies par une variable de type **GLuint**. Pour notre jeu et pour nos balles, nous utilisons une variable nommée **txtballe**.

On commence donc cette fonction par le chargement de l'image d'une balle et de la création d'une surface de même dimension. Nous avons besoin de cette nouvelle surface pour mettre en place le canal alpha. En effet, si vous vous souvenez de l'image d'une balle :



La balle

Cette image est carré mais nous voulons une image ronde lors du rendu. Comment faire ? La réponse est le canal alpha. Rappelons, qu'avec la bibliothèque SDL, on utilise la fonction **SDL\_SetColorKey**. Cette fonction permet de définir la couleur transparente de l'image.

Sous OpenGL, on utilise la valeur du canal alpha pour définir quel pixel sera dessiné ou non. Voyons la suite de cette fonction. Pour regarder et modifier les pixels des surfaces, il suffit d'utiliser les pointeurs **pixels** et **tmppixels**.

#### La suite de la fonction

```

pixels = (unsigned char*) balle->pixels;
tmppixels = (unsigned char*) tmpballe->pixels;

pcnt = 0;
cnt = 0;

//Pour chaque pixel
for(i=0;i<tmpballe->w;i++)
{
    for(j=0;j<tmpballe->h;j++)
    {
        //On copie la couleur
        pixels[pcnt] = tmppixels[cnt];
        pixels[pcnt+1] = tmppixels[cnt+1];
        pixels[pcnt+2] = tmppixels[cnt+2];

        //La couleur magenta sera transparente
        if((tmppixels[cnt]==255)&&(tmppixels[cnt+1]==0)&&(tmppixels[cnt+2]==255))
        {
            pixels[pcnt+3] = 255;
        }
        else
        {
            pixels[pcnt+3] = 0;
        }

        cnt += 3;
        pcnt += 4;
    }
}

```

Je ne pense pas que ce code soit très difficile à comprendre. On parcourt chaque pixel, si la couleur est magenta, on met le canal alpha à 255 sinon il est mis à 0 (remarquez que normalement on utilise une couleur moins utilisée comme le magenta mais c'est un choix arbitraire).

La seule chose dont il faut faire attention, c'est l'incrémentation de 3 pour le compteur **cnt** et 4 pour le compteur **pcnt**. En effet, pour l'image source on a que trois canaux (donc trois octets par pixel) et pour l'image destination on en a quatre (donc quatre octets par pixel).

La suite de cette fonction gère simplement le chargement de la surface sous forme de texture OpenGL. Nous commencerons par générer un indice pour la texture. Ensuite, nous utiliserons la fonction **glBindTexture** pour dire

que c'est cette texture que nous voulons modifier. Les fonctions **glTexParameter** permettent de dire quels filtres nous allons utiliser (voir [cette partie](#) pour plus de détail). Enfin, la fonction **glTexImage2D** permet de vraiment passer le tableau de l'image. Voici le code associé :

#### Chargement de l'image pour OpenGL

```
//On passe la texture a OpenGL

//Generation d'un indice de texture
glGenTextures(1,&txtballe);

//On choisit la texture et on definit un simple filtre pour la texture
glBindTexture(GL_TEXTURE_2D,txtballe);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);

//Chargement de l'image
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,pion->w,pion->h,0,
            GL_RGBA,GL_UNSIGNED_BYTE,balle->pixels);

//Liberation des surfaces
SDL_FreeSurface(balle);
SDL_FreeSurface(tmpballe);
```

Avant de terminer cette fonction, on va passer à la classe **Objet** l'indice de la texture générée.

#### Passer l'indice à la classe Objet

```
//On met a jour la variable statique txtballe de la classe Objet
Objet::setTxtBalle(txtballe);

return true;
}
```

### 4.2.c - Les autres fonctions d'initialisation

Rien de bien particulier pour ces deux autres fonctions d'initialisation. En effet, la fonction **init** permet d'appeler les deux autres fonctions (**initTextures** que nous avons déjà montré et **initObjets** qui remet le nombre d'objets à zéro en utilisant la fonction **clear** de la classe **vector**).

#### Fonctions d'initialisation

```
//Initialisation du jeu
bool Jeu::init()
{
    return initObjets() && initTextures();
}

bool Jeu::initObjets()
{
    objets.clear();
    return true;
}
```

### 4.2.d - La fonction addObjet

La fonction **addObjet** permet d'ajouter une balle dans le jeu s'il y a de la place dans le tableau. Cette fonction retourne -1 s'il y a eu un problème ou l'indice de l'objet qui vient d'être ajouté si tout s'est bien passé.

#### La fonction addObjet

```
int Jeu::addObjet(Objet &o)
{
    // Si on a encore de la place
    if(getNObjets() < MAX_BALLES)
    {
```

#### La fonction addObjet

```

// On copie l'objet dans la case vide
objets.push_back(o);
//On rend l'indice de l'objet insere
return getNObjets()-1;
}
return -1;
}
    
```

Cette fonction appelle la fonction `getNObjets` qui retourne simplement le nombre d'éléments dans le vecteur `objets` :

#### La fonction getNObjets

```

inline unsigned int Jeu::getNObjets()
{
    return objets.size();
}
    
```

### 4.2.e - Les fonctions d'affichage

Les fonctions d'affichage pour la classe **Jeu** se divisent en deux fonctions. La fonction **affiche** est le point d'entrée pour l'affichage du jeu. Pour le moment, elle appelle simplement la fonction **affObjets**.

#### Fonction d'affichage

```

void Jeu::affiche()
{
    //Afficher les objets
    affObjets();
}

void Jeu::affObjets()
{
    int i;

    //On met en place le test sur la valeur alpha
    glEnable(GL_ALPHA_TEST);
    //On dessine si la valeur alpha est inférieur
    glAlphaFunc(GL_LEQUAL,0.1f);

    //On affiche les objets
    for(i=0;i<getNObjets();i++)
    {
        objets[i].affiche();
    }

    //On desactive le test sur la valeur alpha
    glDisable(GL_ALPHA_TEST);
}
    
```

La fonction **glEnable** permet de mettre en place le test Alpha. La fonction **glAlphaFunc** permet de définir comment on gère le canal alpha. En résumé, on définit une valeur seuil (**0.1f** dans ce cas) et avec **GL\_LEQUAL**, on définit qu'on dessine le pixel si le canal alpha est inférieur à **0.1f**. Si vous vous souvenez de la fonction **initTextures**, on a mis 1 pour les pixels magentas et 0 pour les autres, donc les pixels qui ne sont pas de la couleur magenta seront dessinés.

La fonction **affObjets** permet donc d'afficher tous les objets. Comme vous le voyez, nous entourons la boucle d'instructions permettant de tester le canal alpha. Enfin, vous remarquerez que, comme pour la série sur le morpion, c'est à chaque objet de gérer son affichage.

### 4.2.f - La fonction updateObjets

#### La fonction updateObjets

```
void Jeu::updateObjets()
{
    int i;

    // On met à jour chaque objet
    for(i=0;i<getNObjets();i++)
    {
        objets[i].updatePos();
    }
}
```

Comme pour la fonction d'affichage, cette fonction est le point d'entrée pour la mise à jour des positions des objets. Comme vous le voyez, on parcourt simplement les objets et on appelle la fonction pour mettre à jour leur position.

#### 4.2.g - La fonction gereSceneServeur

La fonction **gereSceneServeur** est la dernière fonction que nous allons présenter ici. C'est une fonction qui va appeler la fonction **updateObjets** et ensuite regarder s'il est temps d'ajouter une nouvelle balle.

#### La fonction gereSceneServeur

```
void Jeu::gereSceneServeur()
{
    int vx,vy;
    int r,g,b;
    Objet tmpobj;

    // On met à jour les objets
    updateObjets();

    // Si on doit ajouter un objet
    if(SDL_GetTicks()>last+1000)
    {
        last = SDL_GetTicks();

        // Definit la position du nouvel objet (milieu de l'ecran)
        tmpobj.setPos(WIDTH/2-TAILLE_BALLE, HEIGHT/2-TAILLE_BALLE);

        // Choisit aleatoirement une direction (refusant un 0 pour vx ou vy)
        vx = Rand::randi(5000) - 2500;
        if(vx == 0)
            vx++;
        vy = Rand::randi(5000) - 2500;
        if(vy == 0)
            vy++;
        tmpobj.setDirVitesse(vx,vy);

        // On definit une norme de 4 pour la vitesse
        tmpobj.setVitesse(4);
        // On definit la taille
        tmpobj.setTaille(TAILLE_BALLE,TAILLE_BALLE);

        // On choisit une couleur avec un minimum de 50 par couleur
        r = Rand::randi(205)+50;
        g = Rand::randi(205)+50;
        b = Rand::randi(205)+50;

        tmpobj.setCouleur(r/256.0f, g/256.0, b/256.0);

        // On ajoute l'objet
        addObjet(tmpobj);
    }
}
```

Comme vous le voyez, on met à jour la position des objets et ensuite on regarde s'il est temps d'ajouter un objet dans le jeu. Si c'est le cas, on crée un objet au milieu de l'écran et on choisit une direction pour la vitesse aléatoirement (par contre, on définit la norme comme étant égale à 4). Ensuite, on choisit aussi une couleur aléatoire. Enfin, on ajoute l'objet dans le jeu.

Pour ne pas avoir trop de balles dans la fenêtre, on définit une variable **MAX\_BALLES** qui permet de limiter le nombre.

Je pense que les noms des fonctions de la classe **Objet** sont assez clairs mais je vais prendre deux phrases pour les expliquer. **setDirVitesse** permet de définir la direction de la balle. **setVitesse**, **setTaille** et **setCouleur** permettent respectivement de définir la vitesse, la taille et la couleur de la balle.

## 5 - La classe Objet

Nous avons déjà présenté la classe **Moteur** et la classe **Jeu**. Nous allons maintenant prendre un moment pour présenter la classe **Objet**.

### 5.1 - Le fichier Objet.h

La classe **Objet** est la base de tout le programme. En effet, c'est la plus petite entité dans le jeu (remarquez que dans cette partie, on parlera encore de classes **Vecteur** et **Rand** mais ce ne sont que des classes pour aider la définition des autres, pas des classes pour le jeu).

#### 5.1.a - Les champs membres

Les champs membres de la classe **Objet** sont définis :

##### Les membres de la classe Objet

```
static GLuint txtballe;

// Position, vitesse et taille de l'objet
Vecteur pos,
vitdir,
taille;

// Couleur
float r,g,b;
```

Comme vous le voyez, il n'y a pas grand chose de spécial dans cette classe. Nous avons une variable statique pour l'indice de la texture pour les balles. Ensuite, nous avons un vecteur pour la position, la direction de la balle et la taille de la balle. Enfin, nous avons trois variables pour la couleur.

Nous n'allons pas présenter chaque fonction de cette classe (il y en a beaucoup pour chaque élément qui doit être accédé ou modifié. Nous allons présenter ceux qui ont un intérêt :

##### Les fonctions membres

```
static void setTxtBalle(GLuint t);

void setPos(double, double);
void setTaille(double, double);
void setVitesse(double);
void setCouleur(float,float,float);

//Fonction d'affichage
void affiche();

//Fonction de gestion de collision
void updatePos();
```

La première fonction **setTxtBalle** est une fonction statique pour définir la valeur du champs statique **txtBalle**. Ensuite, nous avons quatre fonctions pour définir la position, la taille, la vitesse et la couleur de la balle.

Ensuite nous avons une fonction pour afficher la balle et une pour mettre à jour la position.

#### 5.1.b - Le fichier Objet.cpp

Le code des fonctions de la classe **Objet** est assez simple, on va montrer les fonctions les unes après les autres.

On commence par ne pas oublier de mettre la variable statique **txtballe** à zéro.

#### La variable statique

```
//Variable statique
GLuint Objet::txtballe = 0;
```

Ensuite nous avons la fonction qui définit la valeur de cette variable statique :

#### La fonction setTxtBalle

```
//Mise en place de la texture d'une balle
void Objet::setTxtBalle(GLuint t)
{
    txtballe = t;
}
```

Nous allons maintenant présenter les fonctions **set** :

#### Les fonctions set

```
//Mise en place de la position de l'objet
void Objet::setPos(double x1, double y1)
{
    pos.setX(x1);
    pos.setY(y1);
}

//Mise en place de la direction de l'objet
void Objet::setDirVitesse(double vx1, double vy1)
{
    vitdir.setX(vx1);
    vitdir.setY(vy1);
}

//Mise en place de la taille de l'objet
void Objet::setTaille(double w1, double h1)
{
    taille.setX(w1);
    taille.setY(h1);
}
```

Comme vous le remarquez, ces fonctions ne font que passer l'information aux variables de type **Vecteur**.

Il reste la fonction d'affichage et la fonction de mise à jour de la position. Tout d'abord, nous allons voir la fonction d'affichage :

#### La fonction affiche

```
// Fonction d'affichage
void Objet::affiche()
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, txtballe);

    glColor3u(r,g,b);
    glBegin(GL_QUADS);
        glTexCoord2i(0,1);
        glVertex3f(pos.getX(),pos.getY(),0); // En haut a gauche
        glTexCoord2i(1,1);
        glVertex3f(pos.getX()+taille.getX(),pos.getY(),0); // En haut a droite
        glTexCoord2i(1,0);
        glVertex3f(pos.getX()+taille.getX(),pos.getY()+taille.getY(),0); // En bas a droite
        glTexCoord2i(0,0);
        glVertex3f(pos.getX(),pos.getY()+taille.getY(),0); // En bas a gauche
    glEnd();
    glDisable(GL_TEXTURE_2D);
}
```

Le code est assez clair, on met en place l'utilisation des textures. On dit qu'on veut utiliser la texture des balles. On définit la couleur de cette balle et on dessine le quadrilatère de la balle.

La dernière fonction que nous allons présenter de cette classe **Objet** est la fonction **updatePos**. En utilisant la position courante et la vitesse, on peut calculer la nouvelle position. Une fois mise en place, on va tester pour voir si la balle est en train de sortir de l'écran, si c'est le cas, on la remet au bord et on modifie la vitesse pour la faire repartir dans la direction opposée.

#### La fonction updatePos

```
//Fonction de mise a jour
void Objet::updatePos()
{
    //Calcul de la nouvelle position
    double x = pos.getX() + vitdir.getX(),
           y = pos.getY() + vitdir.getY(),
           w = taille.getX(),
           h = taille.getY();

    //Mise a jour
    pos.set(x,y);

    //Verification de la position ie si on sort de l'ecran on "rebondit"
    //A gauche
    if(x<0)
    {
        pos.setX(1);
        if(vitdir.getX()<0)
            vitdir.setX(vitdir.getX()*(-1));
    }

    //A droite
    if(x+w>=WIDTH)
    {
        pos.setX(WIDTH-w-1);
        if(vitdir.getX()>0)
            vitdir.setX(vitdir.getX()*(-1));
    }

    //En haut
    if(y<0)
    {
        pos.setY(1);
        if(vitdir.getY()<0)
            vitdir.setY(vitdir.getY()*(-1));
    }

    //En bas
    if(y+h>=HEIGHT)
    {
        pos.setY(HEIGHT-h-1);
        if(vitdir.getY()>0)
            vitdir.setY(vitdir.getY()*(-1));
    }
}
```

Vous remarquerez le test sur la positivité de la direction. Cela permet d'être sûr qu'on modifie correctement la vitesse lorsque c'est nécessaire.

## 6 - La classe Vecteur

La classe Vecteur est relativement simple, nous présenterons simplement la définition de la classe.

### La classe Vecteur

```
class Vecteur
{
    private:
        double x, y;

    public:
        Vecteur();
        ~Vecteur();

        void set(double, double);
        void setX(double x);
        void setY(double y);
        void normalise();

        void setNormalise(double, double);

        double getX();
        double getY();
};
```

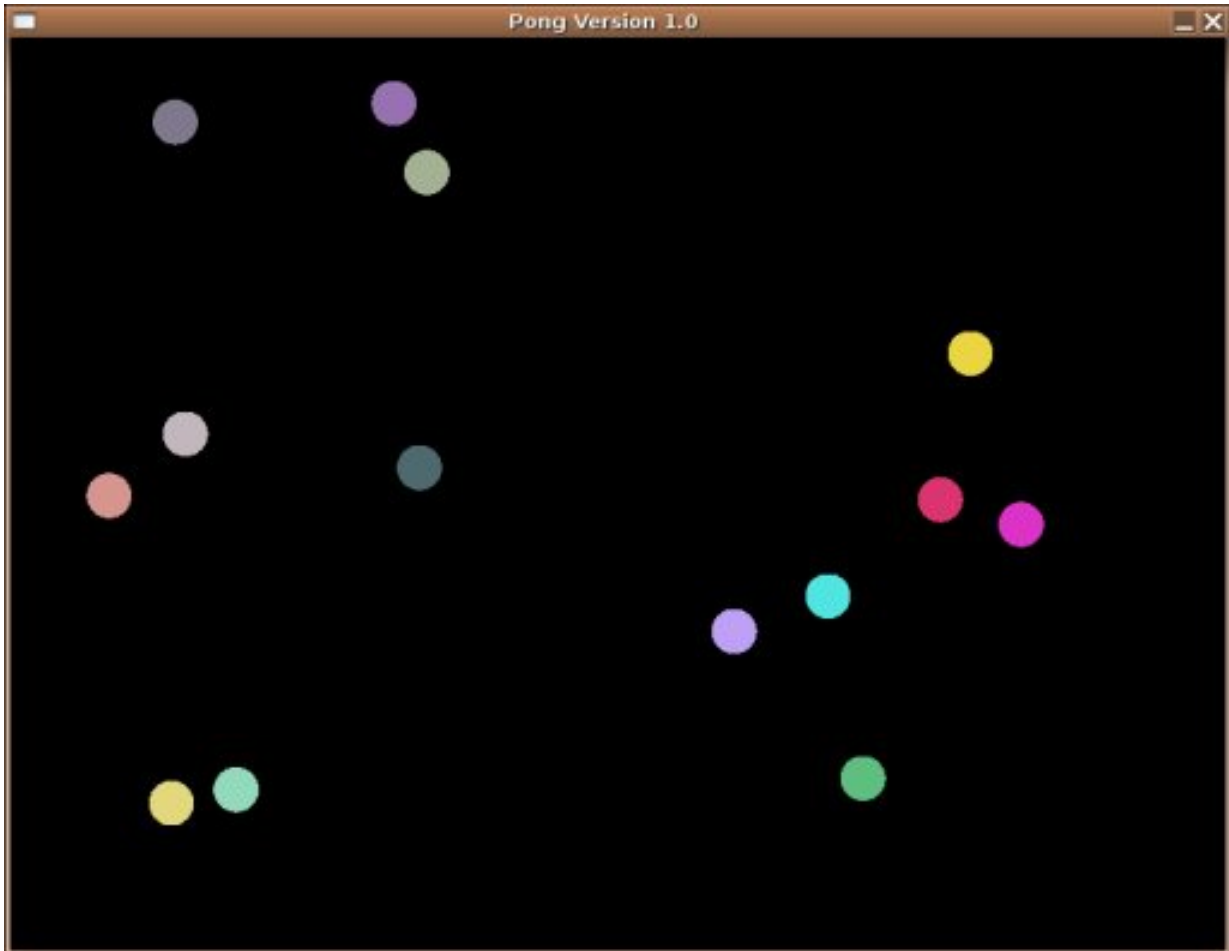
Je ne pense pas que ce soit la peine d'expliquer plus le code. Cette classe a deux membres **x** et **y** et les fonctions permettent de définir la valeur de ces membres. Les seules fonctions intéressantes sont **normalise** qui permet de normaliser le vecteur et la fonction **setNormalise** qui permet de mettre en place le vecteur placé en paramètre et ensuite on normalise le vecteur.

## 7 - La classe Rand

La dernière classe que nous allons présenter est la même que celle qui a été présentée dans la série morpion. Regardez [ici](#) pour plus de détails.

## 8 - Conclusion

Voilà, la deuxième partie d'une série est toujours la plus longue puisqu'on doit prendre le temps de bien présenter toutes les classes qui sont en jeu. Voici une image de ce que nous venons de faire :



*Une image du programme actuel*

La classe `Moteur` permet de faire le lien entre les informations données par la fonction `main` (et donc la bibliothèque SDL) et le jeu sous-jacent. La classe `Jeu` permet de définir et de gérer la cohérence entre tous les objets sous-jacents. Par exemple, le moteur va dire au jeu d'afficher tous les objets.

Bien sûr chaque objet possède aussi son propre indépendance. Ils mettent à jour leur position, savent s'afficher à l'écran.

Ceci conclut donc cette partie du tutoriel. Nous allons montrer comment gérer simplement les collisions dans la prochaine partie.

Jc

- [Sommaire du tutoriel;](#)
- [Introduction;](#)
- [Les bases du moteur;](#)

- [Les collisions et un menu;](#)
- [Améliorer les collisions;](#)
- Le score, la souris et les joueurs;
- Le réseau;
- Conclusion.

## 9 - Téléchargements

Voici le code source pour ce tutoriel: [\(15.6 Ko\) \(version naïve\)](#), [\(15.6 Ko\) \(version SDL\\_framerate\)](#).

Voici la version pdf de cet article: [\(115 Ko\)](#).

Si vous avez des suggestions, remarques, critiques, si vous avez remarqué une erreur, ou bien si vous souhaitez des informations complémentaires, n'hésitez pas à me contacter !

## 10 - Remerciements

J'aimerais remercier [loka](#) pour sa double relecture de cet article !