

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

par [Jean Christophe Beyler](#)

Date de publication : 9/10/2006

Dernière mise à jour : 09/10/2006

Dans cette partie, nous allons nous intéresser aux collisions des balles et à la création d'un menu. Il existe beaucoup de solutions pour cette gestion mais nous allons voir ici une solution simple et facile à mettre en place. Nous allons présenter une classe Physique. Elle sera responsable pour le déplacement des objets, mais aussi des collisions entre les objets. Enfin, nous présenterons une solution rapide pour mettre en place un menu avec des raccourcis claviers.

- 1 - Introduction
- 2 - Les collisions
  - 2.1 - Classe Physique
  - 2.2 - Collision Cercle-Cercle
  - 2.3 - Collision Rectangle-Rectangle
  - 2.4 - La gestion des collisions entre les objets
  - 2.5 - La fonction updateObjets
  - 2.6 - Appel au moteur physique
  - 2.7 - L'insertion des balles
- 3 - Le menu
  - 3.1 - Les raccourcis du clavier
  - 3.2 - Le bouton Continuer
- 4 - La classe Jeu
- 5 - La classe Moteur
- 5 - Conclusion
- 6 - Téléchargements
- 7 - Remerciements

## 1 - Introduction

Bienvenue à cette troisième partie sur la programmation du pong. Dans cette partie, nous allons nous intéresser à la gestion des collisions. La solution proposée ici est assez simple pour être facilement mise en place.

Cette partie présentera comment gérer les collisions et on introduira un menu dans l'application.

## 2 - Les collisions

Nous allons présenter dans cette section la gestion de collisions entre les objets. Ceci va se faire en intégrant une nouvelle classe **Physique**.

### 2.1 - Classe Physique

La classe Physique sera entièrement composée de fonctions statiques. Il existe deux fonctions publiques :

- **updateObjets** : cette fonction met à jour la position et la vitesse des objets (en tenant compte bien sûr des collisions)
- **isCollision** : cette fonction permet de vérifier s'il y a une collision avec le rectangle (x,y,w,h)

Enfin, voici la déclaration de la classe :

#### La classe Physique

```
class Physique
{
    public :
        //Fonction qui met à jour les objets
        static void updateObjets(std::vector<Objet> &objets);

        //Fonction qui teste la collision entre les objets et le rectangle (x,y,w,h)
        static bool isCollision(std::vector<Objet> &objets,
            double x, double y, double w, double h);

    private :
        //Fonction qui gere la collision avec l'objet i
        static int collisionObjet(std::vector<Objet> &objets, int i);
        //Fonction qui teste la collision entre deux rectangles
        static bool collisionRect(double x1,double y1,double w1, double h1,
            double x2, double y2,double w2,double h2);
        //Fonction qui teste la collision entre deux cercles
        static bool collisionCercle(double cx1,double cy1,
            double r1, double cx2, double cy2, double r2);
};
```

Nous allons maintenant voir les fonctions qui gère la collision cercle-cercle et rectangle-rectangle.

### 2.2 - Collision Cercle-Cercle

La collision entre deux cercles est plus facile à gérer. En effet, il suffit de calculer la distance entre deux cercles et voir si elle est inférieure au rayon d'un des cercles. Ce type de collision va être utilisé pour la collision entre deux balles.

#### Collision Cercle-Cercle

```
//Collision entre deux cercles (cx1,cy1,r1) et (cx2,cy2,r2)
bool Physique::collisionCercle(double cx1, double cy1, double r1,
    double cx2, double cy2, double r2)
{
    //Distance entre les deux centres
    double d = (cx1-cx2)*(cx1-cx2) + (cy1-cy2)*(cy1-cy2);
    d = sqrt(d);
    return (d<r1+r2);
}
```

### 2.3 - Collision Rectangle-Rectangle

Il se peut que notre programme ait besoin d'une gestion de collision entre deux rectangles. Dans notre cas, nous aurons besoin de ce type de collision pour gérer la collision entre les balles et la raquette du jeu.

#### Collisions entre deux rectangles

```
//Fonctions qui gerent les collisions
bool Physique::collisionRect(double x1, double y1, double w1, double h1,
                             double x2, double y2, double w2, double h2)
{
    if(x1+w1<x2) return false;
    if(x2+w2<x1) return false;
    if(y1+h1<y2) return false;
    if(y2+h2<y1) return false;

    return true;
}
```

Comme vous le voyez, ce n'est pas très compliqué. La question est de savoir si le rectangle (x1,y1,w1,h1) est en collision avec (x2,y2,w2,h2).

On commence par regarder si l'abscisse **x2** est à gauche de **x1+w1**. Si c'est le cas, il peut y avoir une intersection si **x1** n'est pas à droite de **x2+w2**. En effet, cela voudrait dire que soit **x2** appartient à l'intervalle [x1, x1+w1], soit **x1** appartient à l'intervalle [x2,x2+w2]. Dans ce cas, on regarde si y1 est inclu dans [y2, y2+h2] ou si y2 est inclu dans [y1, y1+h1].

En faisant un dessin, je pense que vous arriveriez à voir tous les cas possibles et voir que nous les gérons tous... Ce type de collision sera utilisé par la collision entre une balle et une raquette.

## 2.4 - La gestion des collisions entre les objets

Nous savons à présent gérer les collisions entre les balles et les collisions entre une balle et une raquette. Bien que nous n'aurons pas encore de raquette dans le programme actuel, nous allons préparer le terrain pour son introduction qui sera fait dans la prochaine partie.

On va poser comme hypothèse qu'avant la mise à jour des objets, aucun objet n'est en collision. Lorsque nous bougeons un objet, nous allons vérifier s'il y a eu une collision. Dans le cas d'une collision, nous remettons l'objet en place (donc à un endroit sans collision) et nous modifions sa direction.

Voici la fonction gérant les collisions :

#### Fonction testant la collision

```
//Est-ce que l'objet d'indice idx est en collision avec les autres ?
int Physique::collisionObjet(std::vector<Objet> &objets, int idx)
{
    int i;
    int nobjets = objets.size();

    double x1,x2,y1,y2;
    double w1,h1,w2,h2;

    double cx1=0,
           cy1=0,
           cr1=0,
           cx2=0,
           cy2=0,
           cr2=0;

    //On recupere la position et taille de l'objet idx
    x1 = objets[idx].getX();
    y1 = objets[idx].getY();
    w1 = objets[idx].getW();
    h1 = objets[idx].getH();
}
```

### Fonction testant la collision

```

//Pour le moment on n'a que des balles, on recupere son centre et rayon
cx1 = x1+w1/2;
cy1 = y1+h1/2;
cr1 = w1/2;

//On parcourt tous les autres objets
for(i=0;i<nobjets;i++)
{
    if(idx!=i)
    {
        x2 = objets[i].getX();
        y2 = objets[i].getY();
        w2 = objets[i].getW();
        h2 = objets[i].getH();

        //Recuperation du deuxieme centre et rayon
        cx2 = x2 + w2/2;
        cy2 = y2 + h2/2;
        cr2 = w2/2;

        //Si on a une collision entre les deux objets
        if(collisionCercle(cx1,cy1,cr1,cx2,cy2,cr2))
            return i;
    }
}
return -1;
    
```

Cette fonction n'est pas très compliquée. On commence par récupérer la position de l'objet en question. Puisque, pour le moment, nous avons que des balles, on utilise la position, la largeur et hauteur pour calculer le centre du cercle et son rayon.

Ensuite, on parcourt chaque objet et on regarde si on a une collision. Si les deux objets sont des cercles (donc notre cas pour le moment) alors nous devons tester pour une collision cercle-cercle.

Le retour de cette fonction représente l'objet qui est en collision ou -1. Ceci permettra au moteur physique de gérer la position des deux objets et leur vitesse.

## 2.5 - La fonction updateObjets

La dernière fonction que nous allons présenter est nommée **updateObjets**. Cette fonction met à jour la position des objets du jeu. L'algorithme de la fonction met à jour la position de chaque objet puis vérifie s'il y a eu ou non une collision.

Dans le cas d'une collision, on remet l'objet dans sa position initiale et on met à jour la direction de l'objet. Voici l'implémentation de cette fonction, nous commencerons par présenter les déclarations des variables :

### La fonction updateObjets

```

void Physique::updateObjets(std::vector<Objet> &objets)
{
    int i;
    int res;
    double oldx, oldy;

    double vx,vy,newvx, newvy,n;
    double x1,x2,y1,y2;
    
```

Ensuite, nous avons une simple boucle qui parcourt chaque objet. Le programme sauvegarde la position de l'objet avant de la mettre à jour. Ensuite, on vérifie s'il y a eu une collision :

### Mise a jour de l'objet

```

for(i=0;i<nobjets;i++)
{
    //Sauvegarde de l'ancienne position
    oldx = objets[i].getX();
    oldy = objets[i].getY();

    //Mise a jour de la position
    objets[i].updatePos();

    //Est-ce qu'on a une collision
    res = collisionObjet(objets, i);
}

```

Une fois le calcul de collision a été fait, on peut regarder la valeur de **res** pour une éventuelle collision. Si nous avons une collision, nous allons remettre la position de l'objet en place.

### Remettre la position de l'objet

```

if(res>=0)
{
    //Si on a une collision, on remet l'ancienne position
    objets[i].setPos(oldx,oldy);
}

```

Maintenant, le code le plus compliqué de ce tutoriel : la gestion de la collision. En effet, nous savons à présent qu'il y a eu une collision entre l'objet courant **i** et l'objet **res**. Il faut maintenant mettre à jour la direction des objets.

### Gestion de la vitesse

```

//Recupere le centre de l'objet i
x1 = oldx + objets[i].getW()/2;
y1 = oldy + objets[i].getH()/2;

//Recupere le centre de l'objet res
x2 = objets[res].getX() + objets[res].getW()/2;
y2 = objets[res].getY() + objets[res].getH()/2;

//Calcul le vecteur de direction entre les centres
vx = x1 - x2;
vy = y1 - y2;

//Normalise le vecteur
n = vx*vx + vy*vy;
n = sqrt(n);

vx /= n;
vy /= n;

```

Comme vous le voyez, le programme commence par calculer le centre des deux objets. On calcule ensuite le vecteur normalisé qui représente la direction de l'objet **res** vers l'objet **i**. Le code qui suit vérifie que l'objet est un cercle (ce qui, pour cette demo, sera toujours le cas) et met la vitesse à jour.

### Mettre à jour la vitesse des objets

```

//Mis a jour de l'objet i
newvx = vx;
newvy = vy;

objets[i].setDirVitesse(objets[i].getVX()+newvx,
                        objets[i].getVY()+newvy);

//Mis a jour de l'objet res
newvx = (-vx);
newvy = (-vy);

objets[res].setDirVitesse(objets[res].getVX()+newvx,
                          objets[res].getVY()+newvy);
}
}

```

Vous remarquerez ma technique pour mettre à jour la vitesse de l'objet. Ce n'est pas politiquement correct mais je préfère garder le code simple. Une somme de vitesses suffira pour ce programme.

## 2.6 - Appel au moteur physique

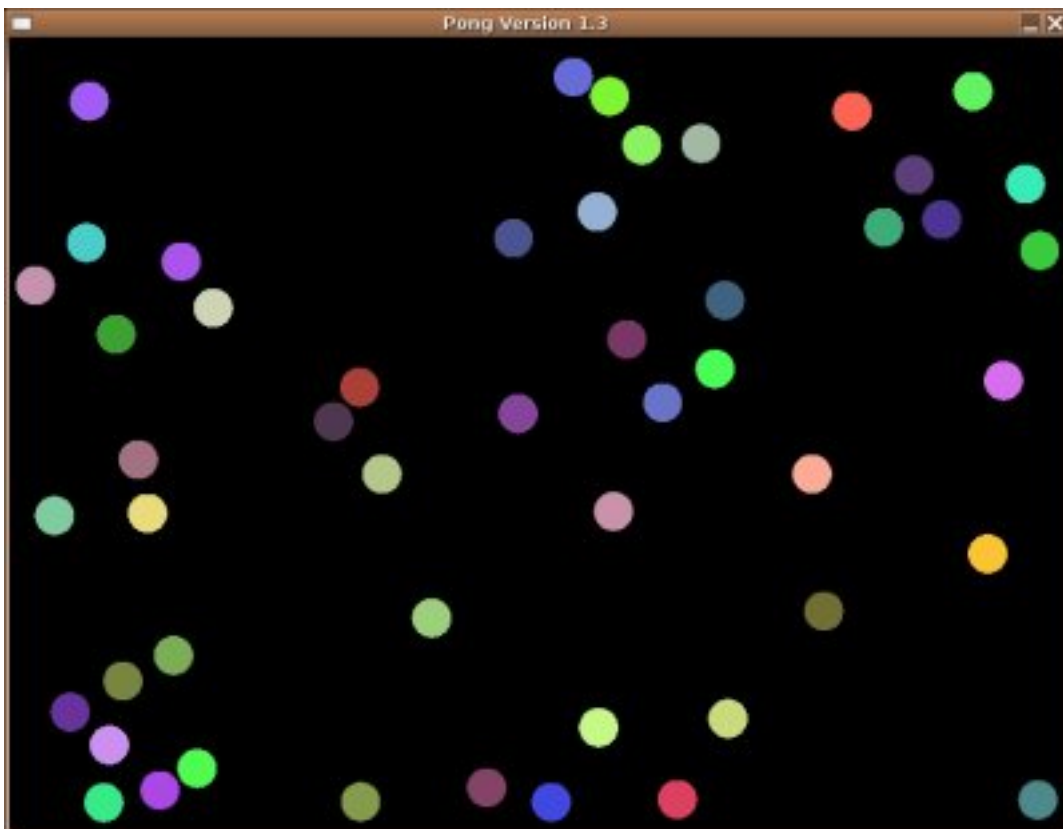
La fonction **gereSceneServeur** va faire un appel au moteur physique passant les informations des objets et du nombre d'objets.

### Appel du moteur physique

```
//Mis a jour des objets
Physique::updateObjets(objets);
```

Comme vous le voyez, la position et de la vitesse des objets sont gérées par le moteur physique et plus directement par la classe **Jeu**. Ceci permet d'avoir une moteur physique indépendant des objets sous-jacents.

Enfin, voici une image de notre pong avec 50 balles:



## 2.7 - L'insertion des balles

La dernière chose à remarquer en rapport avec les collisions concerne l'insertion de nouvelles balles. En effet, pour être sûr qu'une balle puisse être inséré et que cela ne provoquera pas une collision, il suffit d'ajouter ceci dans la fonction **gereSceneServeur** de la classe **Jeu**.

### Test de collision pour la nouvelle balle

```
// Si on doit ajouter un objet
```

#### Test de collision pour la nouvelle balle

```
if((SDL_GetTicks())>last+1000)
    &&(!Physique::isCollision(objets, WIDTH/2-TAILLE_BALLE,HEIGHT/2-TAILLE_BALLE,
        TAILLE_BALLE,TAILLE_BALLE))
```

### 3 - Le menu

Comme dans la série du morpion, nous allons mettre un menu en place. Certes, ce sera juste un menu avec deux boutons. Par contre, nous allons afficher le menu entièrement avec OpenGL pour ne pas mélanger le code d'affichage OpenGL et la bibliothèque SDL.

Cette petite différence est tout de même de taille puisqu'il faut gérer le chargement des fichiers images pour le menu et les faire passer en textures **OpenGL**. Ce détail nous oblige à changer les dimensions de nos images pour qu'elles soient égales à une puissance de deux.

Mais sinon, le code est presque le même, l'affichage du menu ne contient rien de spécial et le code du moteur ressemble énormément au code présenté dans [la partie 5](#) de la série sur le morpion.

Voici une image de notre menu :



#### 3.1 - Les raccourcis du clavier

Il y a tout de même un ajout à ce menu qui le rend nettement plus interactif que celui pour le morpion. Nous avons ajouté des raccourcis clavier (la touche 'n' pour *Nouvelle partie* et la touche 'q' pour *Quitter*).

Ceci se fait facilement avec une fonction **clavier** qui retourne si oui ou non la touche a servie.

#### Gestion du clavier

```
//Gestion du clavier
bool Menu::clavier(unsigned char k)
{
    switch(k)
    {
        case 'c':
            if(moteur.jeuEnCours()) {
                moteur.echangeFonctions();
            }
            return true;
        case 'n':
            moteur.initJeu();
            moteur.setFonctionsJeu();
            return true;
        default:
            return false;
    }
}
```

Et là on remarque qu'un raccourci n'est pas présent lors de l'affichage du menu ! Et nous avons un appel à une fonction **jeuEnCours**. Comme son nom l'indique nous avons une fonction membre de la classe Moteur qui permet de dire si le jeu est en cours. Si le jeu est en cours, nous avons en fait un bouton supplémentaire.

### 3.2 - Le bouton Continuer

Ce bouton *Continuer* apparaît seulement lorsque la fonction membre **jeuEnCours** retourne vrai. En effet, si nous regardons la fonction d'affichage du menu, on remarque que l'affichage se fait en utilisant les fonctions OpenGL et donc rien ne devrait être surprenant :

#### Affichage du menu

```
//Fonction d'affichage
void Menu::affiche()
{
    int i;
    glEnable(GL_TEXTURE_2D);
    glDisable(GL_DEPTH_TEST);

    glAlphaFunc(GL_LESS,10);
    glEnable(GL_ALPHA_TEST);

    glColor3f(1.0,1.0,1.0);

    //On dessine les diffentes images
    for(i=0;i<4;i++)
    {
        glBindTexture(GL_TEXTURE_2D, images[i]);
        glBegin(GL_QUADS);
            glTexCoord2i(0,0);glVertex2i(positions[i].x,positions[i].y);
            glTexCoord2i(1,0);glVertex2i(positions[i].x+positions[i].w,positions[i].y);
glTexCoord2i(1,1);glVertex2i(positions[i].x+positions[i].w,positions[i].y+positions[i].h);
            glTexCoord2i(0,1);glVertex2i(positions[i].x,positions[i].y+positions[i].h);
        glEnd();
    }

    //Pour le dernier bouton, cela dépend si le jeu est en cours
    //Remarquez i vaut 4, donc c'est bien
    if(moteur.jeuEnCours())
    {
        glBindTexture(GL_TEXTURE_2D, images[i]);
        glBegin(GL_QUADS);
            glTexCoord2i(0,0);glVertex2i(positions[i].x,positions[i].y);
            glTexCoord2i(1,0);glVertex2i(positions[i].x+positions[i].w,positions[i].y);
glTexCoord2i(1,1);glVertex2i(positions[i].x+positions[i].w,positions[i].y+positions[i].h);
            glTexCoord2i(0,1);glVertex2i(positions[i].x,positions[i].y+positions[i].h);
        glEnd();
    }
}
```

#### Affichage du menu

```
} glDisable(GL_TEXTURE_2D);
```

Comme vous le voyez, nous avons encore fait un cas particulier pour l'affichage du bouton *Continuer*. Si le jeu est en cours, nous l'affichons, sinon il n'apparaît pas.

La fonction **jeuEnCours** appelle simplement la fonction **enCours** de l'instance du jeu. La classe **Jeu** possède un booléen qui représente si une partie est en cours. Si c'est le cas, ce booléen vaut vrai sinon il vaut faux. Comme toujours, la classe Moteur sert de lien entre le menu et le jeu.



## 4 - La classe Jeu

Dans la classe **Jeu**, il n'y a pas eu beaucoup de changements par rapport au deuxième tutoriel. Un des grands changements est la perte de la mise à jour des objets. En effet, cela se fait maintenant lors de l'appel au moteur Physique.

Il y a toutefois un ajout, la gestion des raccourcis clavier :

```
//Gestion du clavier
bool Jeu::clavier(unsigned char k)
{
    switch(k)
    {
        //On veut voir le menu
        case 'q':
            moteur.echangeFonctions();
            return true;
        default:
            return false;
    }
}
```

Comme nous le voyons ici, seule la touche 'q' sert pour passer du jeu au menu. Comme la fonction **clavier** de la classe **Menu**, cette fonction retourne un booléen exprimant si la touche a été gérée ou non. Ceci permettra au moteur de savoir s'il doit encore gérer la touche (si nécessaire) ou non.

Cette méthode nous permet donc de définir des touches que le jeu sous-jacent (qu'on soit dans le menu ou dans le jeu) peut décider de gérer ou laisser au moteur leur comportement par défaut.

Par exemple, nous pouvons supposer que par défaut la touche 'q' permet de quitter le programme. Dans ce cas, si nous sommes dans le jeu, nous voulons d'abord passer par le menu. Donc nous ajoutons un cas 'q' dans la fonction **clavier** de la classe **Jeu**. En retournant **true**, la classe **Jeu** dit "C'est bon, j'ai géré la touche".

Pour continuer cet exemple, vous avez vu que dans le menu, nous ne gérons pas la touche 'q'. Nous faisons ceci parce que par défaut, si une touche 'q' est appuyé, le moteur quittera le programme **si** l'événement n'est pas géré par le menu ou le jeu. Voici le code qui gère le clavier dans la classe **Moteur** :

```
void Moteur::clavier(unsigned char k)
{
    int gere=false;
    if(dans_menu)
    {
        gere = menu->clavier(k);
    }
    else
    {
        gere = jeu->clavier(k);
    }

    //Si la touche n'est pas geree
    if(!gere) {
        switch(k)
        {
            //On veut voir le menu
            case 'q':
                fin();
                break;
            default:
                break;
        }
    }
}
```



## 5 - La classe Moteur

La classe **Moteur** contient quelques nouvelles fonctions :

### Les nouvelles fonctions de la classe Moteur

```
//Gestion du clavier
void clavier(unsigned char k);

//Est-ce que la partie est finie ?
bool estFini();
//Terminer la partie
void fin();

//Echange entre menu et jeu
void echangeFonctions();

//Initialise la partie
void initJeu();

//Est-ce que le jeu est en cours
bool jeuEnCours();
```

La plupart des fonctions sont assez claires. Le moteur est maintenant capable de dire si le programme sous-jacent se termine. On peut échanger d'état entre la partie et le menu.

La fonction **initJeu** permet de demander au jeu de recommencer la partie. Ceci permet au menu de faire savoir qu'on veut une nouvelle partie. Enfin, la fonction **jeuEnCours** nous fait savoir si une partie est en cours ou si elle n'a pas commencé ou si elle est terminée.

Nous avons aussi introduit la fonction **fin** qui permettra au moteur de faire le nécessaire côté sauvegarde, affichage (un bouton *Etes-vous sûr?* par exemple) et nettoyage.

Ceci modifie bien sûr la fonction **main**. En effet, à la place d'avoir une variable booléenne **done**, nous utilisons maintenant la fonction **fin** pour signaler la fin du programme dans la boucle événementielle et nous utilisons la fonction **estFini** pour savoir si nous devons continuer le programme.

## 5 - Conclusion

Ceci conclut cette troisième partie sur un simple jeu comme un pong. Comme vous le remarquez, ajouter un pseudo-moteur physique n'est pas vraiment compliqué si nous prenons le soin de mettre en place certaines hypothèses.

En effet, en comparant les positions de chaque objet, il est possible d'en déduire les collisions. Ensuite, nous remettons l'objet en place et nous modifions la vitesse de chaque objet. Cette technique, ne suivant pas entièrement les lois physiques suffit pour ce genre de petit programme et est une bonne introduction au sujet.

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.

## 6 - Téléchargements

Voici le code source pour ce tutoriel: [\(781 Ko\) zip](#).

Voici la version pdf de cet article: [\(248 Ko\)](#).

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 !

## 7 - Remerciements

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