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

par Jean Christophe Beyler

Date de publication : 11/01/2007

Dernière mise à jour : 11/01/2007

Nous n'avons jamais parlé de performances depuis le début de ce tutoriel donc je pense qu'il est temps que nous nous y attardions un peu. En faisant des tests de base, on voit qu'on perd beaucoup de temps dans la gestion des collisions. Cette partie va donc discuter d'une solution pour limiter cette perte et rendre notre code de gestion des collisions plus intelligent.

1 - Introduction
2 - Les collisions : version améliorée
  2.1 - Motivations
  2.2 - Présentation
  2.3 - Algorithme
    2.3.a - Structure de données
    2.3.b - Calcul de zone
      2.3.b.1 - La fonction enleverObjetDesZones
      2.3.b.2 - La fonction recupereCoinObjet
      2.3.b.3 - Les fonctions recupereZoneColonne et recupereZoneLigne
      2.3.b.4 - Les fonctions enleverObjetZone
      2.3.b.5 - Les fonctions pour ajouter un objet dans une zone
    2.3.c - Le calcul de collision
    2.3.d - La fonction updateObjets
    2.3.d - La fonction init
  2.4 - Performances
  2.5 - Allocation dynamique
3 - La vitesse des balles
4 - Les barres
  4.1 - Chargement de la texture
  4.2 - Affichage de l'objet
  4.3 - Gestion de la collision
5 - Conclusion
6 - Téléchargements
7 - Remerciements

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

1 - Introduction

Bienvenue à cette quatrième partie sur la programmation du pong. Dans cette partie, nous allons revenir sur la gestion des collisions dans le but de limiter son impact au niveau des performances.

Cette partie présentera donc une technique inspirée des QuadTrees pour ne tester qu'une partie des objets (ceux qui se trouvent dans les environs). Et, à la fin de ce tutoriel, nous allons introduire un nouveau type d'objet : la barre.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

2 - Les collisions : version améliorée

2.1 - Motivations

Il est vrai que dans un jeu de pong, nous n'avons pas forcément besoin de chercher très loin ou d'utiliser un code ultra-performant. Néanmoins, il est souvent intéressant de trouver une meilleure technique qui pourra s'utiliser dans un projet plus complexe.

Question complexité, le code présenté dans la troisième partie n'était pas génial. En effet, si nous supposons que nous avons 50 objets dans le jeu, il fallait faire 2500 comparaisons. Cela ne semblera pas forcément énorme et pour étudier l'impact, j'ai fait quelques tests.

En utilisant le code de la troisième partie et en modifiant la taille de la fenêtre (la passant à 1024*768) et la taille des balles (passant de 30 à 20), voici un tableau récapitulatif du nombre d'images par seconde :

Nombre d'objets	Avec	Sans
50	490	490+
250	104	384
500	30	304

La colonne **Avec** est le nombre d'images par seconde en utilisant la gestion des collisions et la colonne **Sans** est le nombre sans cette gestion.

Comme vous le voyez, la gestion de collision demande tout de même beaucoup de ressources et ralentit tout de même le programme. Bien que cela reste raisonnable pour le cas des 50 objets, je pense que cela est plutôt dû au fait que la carte graphique et l'écran n'arrivent pas à afficher assez rapidement les images (d'où le + dans le tableau). Il faudra donc garder en tête ce problème de saturation dans la suite de l'article. Il faut aussi garder en tête que l'utilisation en force brute de la gestion de collision avec 500 objets nécessite le calcul de 500*500 calculs de collision (ce qui est égal à 250000 calculs...) par mise à jour de la position des objets.

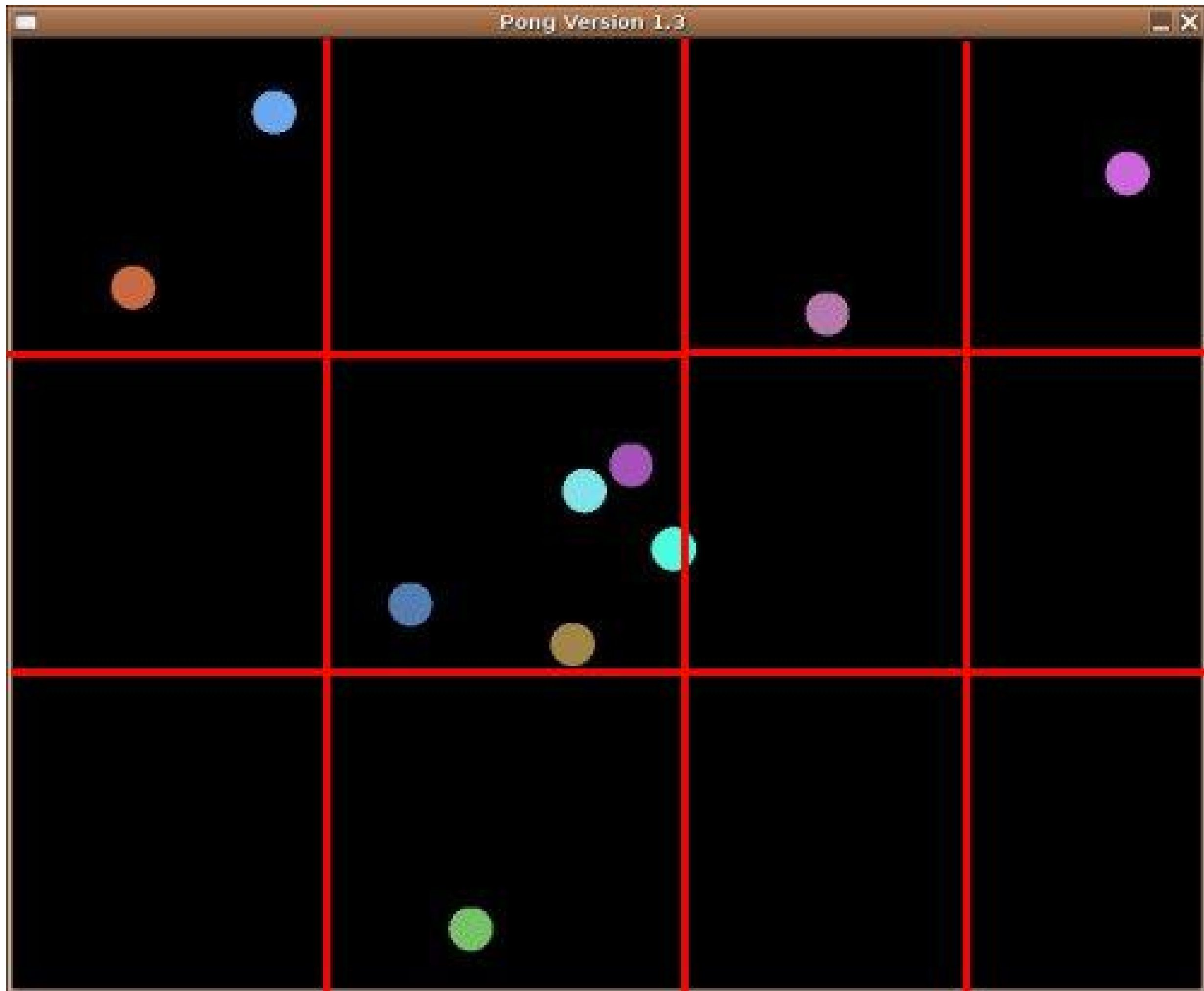
Bien sûr, il faut relativiser l'importance de l'optimisation. Le problème, c'est que 250 objets qui sont en mouvement n'est pas forcément quelque chose d'anormal dans un grand jeu RTS et gérer les collisions ou les interactions entre les objets sont des choses primordiales. Si avec cette gestion, le nombre d'images par seconde est à 100, alors avec un affichage plus complexe qu'un pong, le jeu tournera à 1 ou 2 images par seconde...

2.2 - Présentation

L'idée qui va nous permettre de gagner en nombre d'images par seconde et donc limiter les calculs de collisions est de classer les objets avec, comme facteur de tri, leur position dans le monde de jeu (dans notre cas, c'est donc la position dans la fenêtre).

Sachant bien sûr que nous allons donc tester les collisions d'un objet seulement avec les objets de sa zone. Voici une image montrant la fenêtre avec 10 balles :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



Un quadrillage possible

Le quadrillage montre que, par exemple, dans la zone en haut à gauche, seules deux balles sont potentiellement en collision. Lorsqu'on gérera une de ces balles, ce sera inutile de tester la collision avec les autres.

Par contre, que faire de la balle turquoise qui est sur la ligne ? Et bien, elle se retrouvera dans les deux zones en question. Dernière remarque, rien n'oblige d'avoir le même nombre de lignes que de colonnes dans le quadrillage.

2.3 - Algorithme

Pour arriver à faire cette définition de zones, nous avons donc besoin de pouvoir calculer la zone dans laquelle se trouve un objet et connaître les autres objets de cette zone. Puisque nous déplacerons un seul objet à la fois, l'algorithme général de la mise à jour des objets deviendra donc :

Algorithme général

```
Enlever sa présence des zones auxquelles il appartient
Mis à jour de l'objet
Calcul de ses nouvelles zones
Calcul de collision avec les objets de ces zones
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

Bien entendu, la zone ne changera pas forcément à chaque mise à jour mais il faut bien savoir si nous avons changé de zone, donc il faut le vérifier à chaque mouvement.

2.3.a - Structure de données

Pour définir les structures de données nécessaires pour cet algorithme, nous allons introduire de nouvelles constantes dans le programme :

Constantes pour le quadrillage
```
const int QUADCOL = 5;
const int QUADROW = 5;
```

Nous avons donc défini un quadrillage de 5 colonnes et 5 lignes. Ensuite, pour gérer les objets de notre classe physique, nous devons ajouter un tableau à deux dimensions pouvant contenir ces objets.

Il faut remarquer que la taille minimale des zones est la moitié de la taille d'une balle + 1.

Puisqu'il est possible que tous les objets soient dans la même zone, nous devons avoir un tableau qui peut stocker tous les objets dans la même zone. Nous commencerons par montrer une allocation statique (plus simple d'un point de vue conception) définissant une structure qui contient le nombre d'objets dans la zone et un tableau contenant les numéros des objets.

Structure de zone
```
typedef struct sZone
{
int nobjets;
int objets[MAX_BALLES];
}SZone;
```

Ensuite nous pouvons déclarer, dans notre classe physique, notre tableau de zones. Remarquons que ceci déclare la taille maximale possible et ceci peut être amoindri si on utilise une allocation dynamique. Par contre, dans le cas d'un quadrillage de 5*5 avec 250 objets au maximum et 4 octets par entier, cette table utilisera approximativement 25k ce qui est raisonable.

Tableau de zones
```
//Tableau de zones
SZone tableau_zones[QUADLIGNE][QUADCOL];
```

2.3.b - Calcul de zone

Dans la suite logique des choses, nous devons maintenant pouvoir calculer la zone de chaque objet. En fait, pour ce calcul, il suffira de calculer la zone des quatre coins de l'objet.

Nous aurons donc deux fonctions, une qui demande d'enlever un objet des zones et une fonction qui ajoute un objet aux zones. Voici le code qui enlève un objet donné du tableau des zones.

2.3.b.1 - La fonction enleverObjetDesZones

Enlever un objet des zones
```
void Physique::enleverObjetDesZones(Objet *objs, int idx)
{
      double x=0,y=0;
      int oldzonei=-1, oldzonej=-1;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

Enlever un objet des zones
```
        int newzonei,newzonej;
        //Pour chaque coin du rectangle
        for(int i=0;i<4;i++)
        {
              //Recupere le coin
              recupereCoinObjet(objs[idx],i,x,y);

              //Calcul de la zone
              newzonei = recupereZoneLigne(y);
              newzonej = recupereZoneColonne(x);

              //Si c'est une autre zone
              if((newzonei!=oldzonei)||(newzonej!=oldzonej))
                    {
                          oldzonei = newzonei;
                          oldzonej = newzonej;
                          enleverObjetZone(idx,newzonei,newzonej);
                    }
        }
  }
```

Il y a bien sûr beaucoup de nouvelles fonctions dans ce code. Nous allons les voir dans l'ordre après avoir expliqué l'idée générale de la fonction.

Tout d'abord, nous allons étudier chaque coin de l'objet. Pour connaître la position de chaque coin, nous allons appeler la fonction **recupereCoinObjet**. Pour une question d'optimisation, cette fonction se trouve dans la classe **Physique**. On pourrait la mettre dans la classe **Objet** mais on perdrait la possibilité de mettre le code en inline (la plupart des compilateurs ne font pas d'inline entre fichiers...).

Une fois que nous avons la position du coin, on utilisera les fonctions **recupereZoneLigne** et **recupereZoneColonne** qui permettent respectivement d'avoir la ligne et la colonne correspondante pour le tableau des zones.

Le test qui suit nous permet de ne pas demander le retrait d'un objet si, d'un coin à un autre, nous sommes restés dans la même zone. Il est possible que nous demanderons tout de même deux fois le retrait d'une même zone mais cela est mieux que 4 fois!

Bref, présenté comme cela, cette fonction est assez directe mais en fait elle utilise 4 autres fonctions pour faire son travail. Présentons rapidement la fonction qui nous donne la position du coin.

2.3.b.2 - La fonction recupereCoinObjet

La fonction recupereCoinObjet
```
  inline void Physique::recupereCoinObjet(Objet &o, int i, double &x, double &y)
  {
        //On regarde chaque coin
        switch(i)
        {
              case 0:
                    //En haut a gauche
                    x = o.getX();
                    y = o.getY();
                    break;
              case 1:
                    //En haut a droite
                    x = o.getX()+o.getW();
                    y = o.getY();
                    break;
              case 2:
                    //En bas a droite
                    x = o.getX()+o.getW();
                    y = o.getY()+o.getH();
                    break;
              case 3:
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La fonction recupereCoinObjet
```
                        //En bas a gauche
                        x = o.getX();
                        y = o.getY()+o.getH();
                        break;
                default:
                        break;
        }

}
```

Comme vous le voyez, le paramètre **i** permet de définir quel coin nous intéresse. Ensuite, en utilisant un switch, la fonction est capable de mettre à jour les paramètres **x** et **y**.

2.3.b.3 - Les fonctions recupereZoneColonne et recupereZoneLigne

Les fonctions recupereZoneColonne et recupereZoneLigne
```
inline int Physique::recupereZoneColonne(double x)
{
        const int PIXPARCOL = WIDTH/QUADCOL;
        int res = (int) (x/PIXPARCOL);
        if(res>=QUADCOL)
                {
                return QUADCOL-1;
                }
        return res;
}

inline int Physique::recupereZoneLigne(double y)
{
        const int PIXPARLIGNE = WIDTH/QUADLIGNE;
        int res = (int) y/PIXPARLIGNE;
        if(res>=QUADLIGNE)
                {
                return QUADLIGNE-1;
                }
        return res;
}
```

Ces deux fonctions sont relativement simples si vous vous souvenez de l'image du quadrillage. Le but est de savoir quel indice représente la zone d'une position en pixel. Si nous avons une largeur de 800 et 5 colonnes, chaque colonne fait donc 160 pixels. C'est par cela que commence ces fonctions : le calcul de la largeur (ou hauteur) d'une zone.

Ensuite, il suffit de calculer à quelle zone appartient un objet par une simple division et le test final permet de récupérer les objets qui sont aux extrêmes de l'écran (en effet, un effet de bord du calcul par entier, on perd un peu de précision et il y a donc un risque de débordement).

2.3.b.4 - Les fonctions enleverObjetZone

Nous arrivons enfin à la fonction qui va enlever d'une zone l'indice d'un objet :

La fonction enleverObjetZone
```
void Physique::enleverObjetZone(int idx, int i, int j)
{
   int k;
   int *ptr = tableau_zones[i][j].objets,
       size = tableau_zones[i][j].nobjets;

   for(k=0;k<size;k++)
   {
      if(ptr[k]==idx)
         {
         //On efface en echangeant avec le dernier
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

La fonction enleverObjetZone
```
            ptr[k] = ptr[size-1];
            tableau_zones[i][j].nobjets--;
            break;
            }
        }
    }
```

Ce code est assez simple : on parcourt le tableau représentant la zone en cherchant l'indice de l'objet à enlever. Si on le trouve, on écrase la case avec le dernier objet du tableau et on décrémente le nombre d'objets dans la zone.

Nous avons enfin présenté chaque fonction qui sert à enlever un objet des zones auxquelles il appartient. Comme vous le voyez, il n'y a rien de particulier et rien de très difficile. Il suffit de faire un peu attention aux indices...

2.3.b.5 - Les fonctions pour ajouter un objet dans une zone

Pour ajouter un objet, nous faisons les mêmes opérations sauf qu'à la place d'appeler **enleverObjetZone**, nous allons appeler une fonction **ajouterObjetZone**. Pour garder la longueur de ce tutoriel convenable, je ne vais pas mettre le code (surtout qu'il n'y a pas grand chose de compliqué dans le code !).

2.3.c - Le calcul de collision

Il faut maintenant changer la fonction **collisionObjet** pour qu'elle prenne en compte l'utilisation des zones. Pour garder les deux fonctions, nous nommerons la nouvelle fonction **collisionObjetZone** et voici son prototype :

Le prototype de la fonction collisionObjetZone
```
    //Fonction qui gere la collision avec l'objet i mais en utilisant les zones
    static int collisionObjetZone(std::vector<Objet> &objets, int idx);
```

Cette fonction prend le tableau des objets et l'indice de l'objet à considérer. Puisque nous n'avons plus besoin de parcourir le tableau, nous ne passons plus sa taille. Pour ne pas faire trop long, nous présenterons seulement la partie la plus intéressante de cette fonction (en gros, nous omettons la déclaration des variables et le dernier return) :

La fonction collisionObjetZone
```
        int oldzonei=-1, oldzonej=-1;

        x1 = objets[idx].getX();
        y1 = objets[idx].getY();
        w1 = objets[idx].getW();
        h1 = objets[idx].getH();

        //Si c'est un cercle, on recupere son centre
        if(objets[idx].getType()==CERCLE)
              {
              cx1 = x1+w1/2;
              cy1 = y1+h1/2;
              cr1 = w1/2;
              cercle = true;
              }

        //Pour chaque coin
        for(i=0;i<4;i++)
        {
              //Recupere le coin
              recupereCoinObjet(objets[idx],i,x,y);

              //Calcul de la zone
              newzonei = recupereZoneLigne(y);
              newzonej = recupereZoneColonne(x);

              //Si c'est une autre zone
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La fonction collisionObjetZone

```
                if((newzonei!=oldzonei)||(newzonej!=oldzonej))
                        {
                                oldzonei = newzonei;
                                oldzonej = newzonej;

                                //Ok on va regarder dans cette zone
                                nobjets_zone = tableau_zones[newzonei][newzonej].nobjets;
                                objets_zone = tableau_zones[newzonei][newzonej].objets;

                                //On parcourt tous les objets de cette zone
                                for(k=0;k<nobjets_zone;k++)
                                {
                                        l = objets_zone[k];

                                        x2 = objets[l].getX();
                                        y2 = objets[l].getY();
                                        w2 = objets[l].getW();
                                        h2 = objets[l].getH();

                                        //Si on a deux cercles
                                        if( cercle && (objets[l].getType()==CERCLE))
                                        {
                                                cx2 = x2 + w2/2;
                                                cy2 = y2 + h2/2;
                                                cr2 = w2/2;

                                                if(collisionCercle(cx1,cy1,cr1,cx2,cy2,cr2))
                                                        return l;
                                        }
                                        else //Sinon on fait une collision entre rectangles
                                        {
                                                if(collisionRect(x1,y1,w1,h1,x2,y2,w2,h2))
                                                        return l;
                                        }
                                }
                        }
        }
    return -1;
```

Si on comparait avec la version qui n'utilisait pas de zone, on remarquerait que la boucle **k** interne est du même genre que l'originale. Nous avons simplement entouré cette boucle d'un calcul de zone pour chaque coin de l'objet.

Une petite différence est la disparition d'un test pour vérifier qu'on ne calcule pas la collision de l'objet sur lui-même. En effet, l'algorithme demande d'enlever l'objet du tableau de zone **avant** la gestion des collisions. Puisque cet indice n'est plus dans le tableau, le test est inutile.

Plus important est la présence d'un test sur le type de l'objet. En effet, à la fin de ce tutoriel, nous allons avoir un deuxième type d'objet : les barres ! Du coup, nous avons défini une énumération contenant tous les types. Ceci nous permettra de savoir quel genre de calcul de collision est nécessaire, mais aussi le genre de réaction qu'auront les objets lors de collisions.

L'énumération des types

```
    enum Type {CERCLE, BARRE};
```

Ceci rajoute bien sûr un champs membre dans la classe **Objet** et une fonction qui permet de récupérer le type de l'objet.

2.3.d - La fonction updateObjets

Enfin la dernière chose à modifier est la fonction d'entrée **updateObjets** qui s'occupera de la gestion du tableau de zones et d'appeler la nouvelle fonction **collisionObjetZone**. Pour respecter l'algorithme, il suffira d'enlever l'objet du tableau avant la modification de sa position :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Enlever l'indice de l'objet

```
//Sauvegarde de l'ancienne position
oldx = objets[i].getX();
oldy = objets[i].getY();

//Enlever des zones
enleverObjetDesZones(objets, i);

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

Et bien sûr, avant de passer au prochain objet, il ne faut pas oublier de rajouter l'indice de l'objet dans le tableau :

Ajouter l'indice de l'objet

```
//Ajouter dans les zones
ajouterObjetDansZones(objets, i);
```

2.3.d - La fonction init

Notre classe **Physique** contient maintenant des informations sur l'état du jeu et la position des balles. Il faut donc pouvoir initialiser le tableau pour le mettre à zéro. Ceci se fait lors du début du programme ou lorsque l'utilisateur veut recommencer une partie.

La fonction **init** sert donc à mettre chaque case du maillage à zéro :

La fonction init

```
//Fonction d'initialisation
bool Physique::init()
{
int i,j;
for(i=0;i<QUADLIGNE;i++)
  for(j=0;j<QUADCOL;j++)
    {
    tableau_zones[i][j].nobjets = 0;
    }
return true;
}
```

Enfin, l'instance de jeu appellera l'initialisation du moteur physique dans la fonction **recommence** :

La fonction recommence de la classe Jeu

```
//Recommence la partie
void Jeu::recommence()
{
    objets.clear();
    Physique::init();
    jeuEnCours = true;
}
```

2.4 - Performances

Voici le tableau que vous avez vu au début du tutoriel avec une nouvelle colonne. **Zone** donne le nombre d'images par seconde de cette nouvelle version de calcul de collisions.

Nombre d'objets	Avec	Zone	Sans
50	490	490+	490+
250	104	380	384
500	30	240	304

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

Comme vous le voyez, notre nouvelle version permet d'avoir le même comportement que la version sans gestion des collisions jusqu'à 250 objets. Ensuite, pour 500 objets, nous avons tout de même un gain considérable par rapport à la version de base.

Enfin, nous pouvons faire remarquer que, si l'utilisation mémoire n'est pas trop un facteur, passer à un quadrillage 15*15 permet d'avoir le même nombre d'images par secondes avec gestion de collisions qu'une version sans pour un programme qui fait interagir 500 objets ! Ceci en gardant en tête qu'il y a sûrement un facteur de saturation au niveau de la fréquence d'affichage, mais cela est nettement mieux qu'avoir 30 images par secondes...

A la fin de ce tutoriel, il y aura une version du pong qui contiendra cette allocation statique. L'allocation statique a l'avantage d'être plus facile à concevoir, par contre, le gaspillage de mémoire peut souvent être évité en utilisant l'allocation dynamique. C'est ce que nous allons montrer par la suite.

2.5 - Allocation dynamique

Nous avons vu que les performances de ce système dépendent largement de la position des objets mais aussi de la taille du quadrillage. Si le quadrillage est plus fin, la gestion des collisions peut se faire plus rapidement. Par contre, notre utilisation mémoire est tout de même un peu gourmande, donc nous allons présenter ici comment résoudre ce problème en passant par une allocation dynamique.

Bien que ceci ne soit pas nécessaire, cela est tout de même intéressant de pouvoir garder le gain de temps et de limiter l'occupation mémoire. Bien qu'on n'en ait pas parlé, un quadrillage 15*15 avec 500 balles utilise tout de même 450 kO !

Pour limiter ce nombre, nous allons donc directement utiliser un tableau de type **std::vector**. Nous n'aurons plus besoin d'une structure puisque le type **std::vector** contient la méthode **size** qui nous le fournira.

Nouveau tableau de zones

```
//Tableau de zones
static std::vector<int> tableau_zones[QUADLIGNE][QUADCOL];
```

Ensuite, il faut mettre à jour le code pour gérer ce changement de type. Nous commençons par changer la déclaration de la variable **tableau_zones** :

Déclaration du tableau de zones

```
std::vector<int> Physique::tableau_zones[QUADLIGNE][QUADCOL];
```

Ensuite, à la place d'utiliser la variable **nobjets**, nous passons par la méthode **size** dans les fonctions **enleverObjetZone**, **ajouterObjetZone** et **collisionObjetZone**.

Voici le code pour enlever un objet d'une zone :

Enlever un objet d'une zone

```
void Physique::enleverObjetZone(int idx, int i, int j)
{
    int k;
    std::vector<int> &ptr = tableau_zones[i][j];
    int size = tableau_zones[i][j].size();

    for(k=0;k<size;k++)
    {
        if(ptr[k]==idx)
        {
            //On efface en echangeant avec le dernier
            ptr[k] = ptr[size-1];
            ptr.pop_back();
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

Enlever un objet d'une zone

```
                break;
            }
        }
    }
```

Comme vous le voyez, ce code copie le dernier objet du tableau et ensuite utilise la méthode **pop_back** pour diminuer la taille du vecteur.

Pour ajouter un objet dans une zone, on utilise cette nouvelle version de la fonction **ajouterObjetZone** :

Ajouter un objet d'une zone

```
    void Physique::ajouterObjetZone(int idx, int i, int j)
    {
        unsigned int k,
            size = tableau_zones[i][j].size();
        std::vector<int> &ptr = tableau_zones[i][j];

        for(k=0;k<size;k++)
        {
            if(ptr[k]==idx)
            {
                break;
            }
        }

        if((k==size)&&(size<MAX_BALLES))
            {
            ptr.push_back(idx);
            }
    }
```

Cette nouvelle version utilise la fonction **push_back** pour ajouter un élément dans un vecteur. La dernière fonction qui a été modifiée est **collisionObjetZone**, voici la portion de la fonction qui a été modifiée :

Changement dans collisionObjetZone

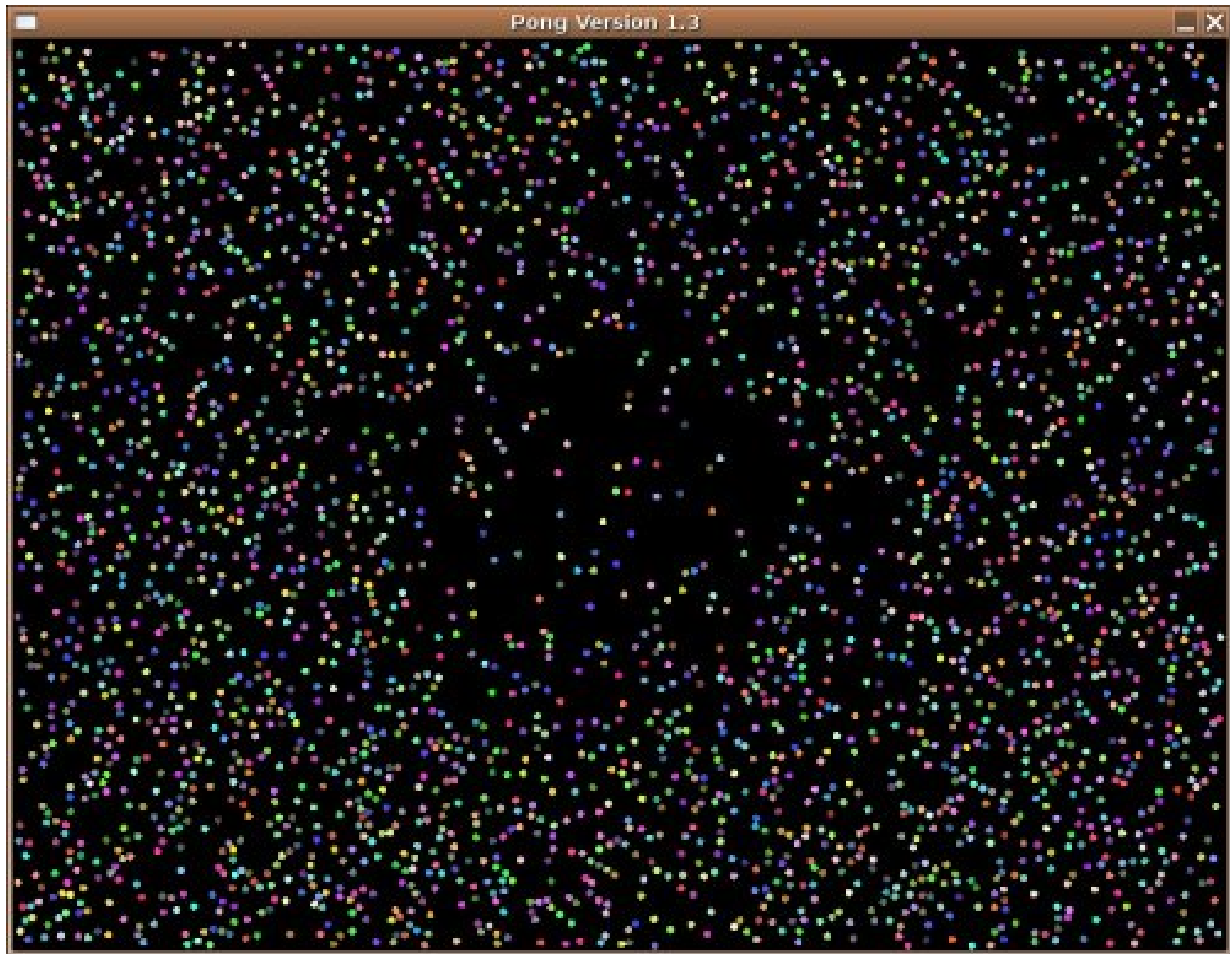
```
        //Ok on va regarder dans cette zone
        nobjets_zone = tableau_zones[newzonei][newzonej].size();
        std::vector<int> &objets_zone = tableau_zones[newzonei][newzonej];

        //On parcourt tous les objets de la zone
        for(k=0;k<nobjets_zone;k++)
            {
            l = objets_zone[k];
```

Ce qui est intéressant est la facilité offerte par le type **std::vector** pour la gestion dynamique de la taille des vecteurs. Ce prochain tableau montre les performances avec 3000 objets de taille 5 (la mémoire est la mémoire utilisée par le programme en méga-octets) :

Version	Mémoire	Taux de rafraîchissement
Gestion naïve	16	2
Statique	60	65
Dynamique	16	68
Sans	16	150

Enfin, voici une image qui montre 3000 balles :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/



3000 balles

Une dernière remarque sur cet algorithme pour dire qu'il ne faut pas non plus prendre n'importe quelle taille pour le quadrillage. En effet, il faut que la largeur (ou hauteur) soit plus grand que le rayon de la plus grande balle. En effet, on regarde les zones des coins des balles. Il faut donc qu'il n'y ait pas de *trou* entre deux coins. Sinon, une collision peut se produire sans que nous la détections. Une bonne solution est de définir les constantes comme ceci :

Définition du quadrillage

```
const int QUADCOL=WIDTH/TAILLE_BALLE;
const int QUADLIGNE=HEIGHT/TAILLE_BALLE;
```

A la fin de ce tutoriel, vous aurez le programme permettant d'avoir les 3000 balles gérées par une allocation statique (puisqu'à la fin de ce tutoriel, vous aurez le source utilisant entièrement l'allocation dynamique mais aussi les barres.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

3 - La vitesse des balles

Pour le moment, la mise à jour de la position de la balle se fait comme ceci :

Code pour la mise à jour des objets
```
//Calcul de la nouvelle position
double x = pos.getX() + vitdir.getX(),
       y = pos.getY() + vitdir.getY(),
       w = taille.getX(),
       h = taille.getY();
```

Comme vous le voyez, il n'y pas de gestion du temps. Ceci n'est pas une très bonne idée puisque, si ce programme est utilisé sur différents ordinateurs, la vitesse de la balle ne sera pas la même. En utilisant le temps, on harmonise les déplacements.

Cette solution n'a pas que des inconvénients : s'il y a une discontinuité dans le taux de rafraîchissement (le système d'exploitation fait autre chose par exemple), le programme continuera comme si de rien n'était. En utilisant le temps, après une discontinuité, il y aura un saut dans le déplacement. Lors d'un calcul de collision, cela n'est pas très pratique. Une solution hybride consiste à calculer la vitesse moyenne de temps en temps, et utiliser cette vitesse pour le déplacement.

En voulant garder le code relativement simple, nous allons tout de même utiliser la version utilisant le temps. Voici le code que nous allons utiliser :

Nouveau code pour la mise à jour des objets
```
//Heure actuelle
long tps = SDL_GetTicks() - derniertps;
//Calcul de la nouvelle position
double x = pos.getX() + vitdir.getX() * tps * RATIOVITESSE,
       y = pos.getY() + vitdir.getY() * tps * RATIOVITESSE,
       w = taille.getX(),
       h = taille.getY();

//Mise à jour derniertps
setDernierTps();
```

En ajoutant une variable **derniertps** qui représente la dernière mise à jour. En utilisant la différence entre le temps actuel et la dernière mise à jour, on est capable de calculer la distance de déplacement. Vous remarquerez la présence de la fonction **setDernierTps** qui permet de mettre à jour la variable **dernierTps**.

La mise en place de dernierTps
```
class Objet
{
  private :
     ...
     //Dernier temps de mise à jour
     long dernierTps;
  pubic :
     ...
     //Mettre a jour la variable derniertps
     void setDernierTps();
```

Le fait que la fonction **setDernierTps** soit déclaré comme une fonction publique n'est pas un hasard. En effet, nous allons avoir besoin de cette fonction lorsque l'utilisateur met le jeu en pause. Avant, nous mettions à jour la position à chaque mise à jour, indépendamment du temps entre les deux.

Cela veut dire que si l'utilisateur met le jeu en pause et revient 15 minutes plus tard. Lorsque le jeu reprend, les balles ne bougeront pas plus rapidement. Par contre, maintenant, vous imaginez bien que la variable **tps** va être trop grande, cela provoquera donc des *sauts*.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Une solution facile est de *toucher* les objets pour mettre à jour leur variable **dernierTps**. Nous déclarons donc une nouvelle fonction **toucheObjets** dans la classe **Jeu** pour les mettre à jour.

La fonction toucheObjets

```
void Jeu::toucheObjets()
{
    unsigned int i,
            nobjets = objets.size();

    for(i=0;i<nobjets;i++)
        {
        objets[i].setDernierTps();
        }
}
```

Cette fonction nous permettra donc de mettre toutes les variables **dernierTps** à jour et donc d'éviter les *sauts*. Bien sûr, il faut encore appeler cette fonction au bon moment. Ceci se fera lorsque le jeu reprend. Nous pouvons le gérer en complétant donc les fonctions de la classe **Moteur** :

La fonction echangeFonctions

```
//Echange entre menu et jeu
void Moteur::echangeFonctions()
{
    dans_menu = !dans_menu;
    if(!dans_menu) {
        jeu->toucheObjets();
    }
}
```

Dernière remarque, ceci modifie bien sûr la norme qui est utilisée pour les vecteurs. En effet, SDL_GetTicks retourne le temps en terme de millisecondes. Ceci veut donc dire qu'une norme correcte serait de l'ordre de 0.1 pour que l'objet se déplace d'une centaine de pixels en une seconde. C'est donc à cela que sert la constante RATIOVITESSE.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

4 - Les barres

La dernière chose (je le promets) que nous allons faire dans ce tutoriel est d'ajouter un nouveau type d'objet. Comme vous l'avez sûrement remarqué nous avions déjà préparé un peu le terrain pour cet objet. En effet, nous avons déclaré cet objet dans l'enum **Type** :

L'enum Type
```
    enum Type {CERCLE, BARRE};
```

Ensuite, si vous vous souvenez de la fonction **collisionObjetZone**, nous avons commencé à intégrer la présence de types d'objets différents. En effet, en fonction du type des objets, nous utilisons la collision la plus adaptée.

Il y a donc un peu de travail nécessaire pour ajouter ce nouveau type. Il faudra gérer son affichage et la réaction qu'il peut y avoir face à une collision.

En ce qui concerne l'affichage, tout sera géré par la classe **Objet**. Il faudra bien sûr charger une nouvelle texture pour nos barres, nous allons ajouter une deuxième variable statique dans la classe **Objet** pour contenir son indice et lors de l'affichage de l'objet, son type décidera entre les deux indices pour l'appel **glBindTexture**.

4.1 - Chargement de la texture

Nous avons donc besoin d'une deuxième variable statique pour la texture :

Le chargement de deux textures
```
    static GLuint txtpion, txtbarre;
```

Pour ce qui est du chargement de la texture, cela se passe dans la fonction membre **initTextures** de la classe **Jeu**. Pour charger les deux textures, nous allons encapsuler le code existant dans une petite boucle qui, grâce à un **if**, chargera d'abord une texture puis la deuxième.

Le chargement de deux textures
```
        for(k=0;k<2;k++)
        {

            if(k==0)
            {
                //On charge l'image "data/balle.bmp"
                tmpsurf = SDL_LoadBMP("data/balle.bmp");
            }
            else
            {
                //On charge l'image "data/barre.bmp"
                tmpsurf = SDL_LoadBMP("data/barre.bmp");
            }

             //Gestion de l'erreur
             if(tmpsurf==NULL)
             {
                   std::cout << "Erreur dans le chargement de la surface : "
                             << SDL_GetError() << std::endl;
                   return false;
             }
```

Ensuite, nous avons le même code que pour la deuxième partie. Par contre, à la fin de la boucle, il faut passer à la classe **Objet** de l'indice de la texture, un autre **if** permet de gérer ce problème :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

Interaction avec la classe Objet

```
            if(k==0)
            {
                //On passe la texture aux objets de type balle
                Objet::setTxtBalle(txtsurf);
            }
            else
            {
                //On passe la texture aux objets de type barre
                Objet::setTxtBarre(txtsurf);
            }
```

A ce point du code, les deux textures sont donc chargées et la classe **Objet** a maintenant les deux variables statiques **txtpions** et **txtbarre** sont à jour.

4.2 - Affichage de l'objet

Lorsque nous allons vouloir afficher l'objet, il faudra simplement regarder le type de l'objet avant de faire l'appel au **glBindTexture**.

Interaction avec la classe Objet

```
    if(type==CERCLE)
       {
       glBindTexture(GL_TEXTURE_2D, txtpion);
       }
    else
       {
       glBindTexture(GL_TEXTURE_2D, txtbarre);
       }
```

Bien sûr, si nous avions plus de deux types, un tableau d'indice de textures OpenGL serait plus approprié.

4.3 - Gestion de la collision

Enfin, pour la gestion de la collision, il suffit de dire que les barres ne sont pas affectées par les collisions et qu'une balle qui est en collision avec la barre repart dans la direction du centre de la barre vers la balle. En calculant la position du centre de la barre, nous avons facilement ce vecteur :

Sachant que pour la collision Cercle-Cercle nous avons déjà calculé cette direction, la seule chose qu'il reste à faire est de ne plus faire la somme des vitesses comme avant :

Collision avec une barre

```
    //Si l'objet res n'est pas un cercle, on prend la direction (newx, newy)
    //et met la vitesse à 0.1
    if(objets[res].getType()!=CERCLE)
       {
       objets[i].setDirVitesse(newvx,newvy);

       }
    else //Sinon on ajoute la direction (newx, newy)
       {
         objets[i].setDirVitesse(objets[i].getVX()+newvx,
              objets[i].getVY()+newvy);
       }
    objets[i].setVitesse(0.1);
```

Donc, lorsqu'une balle rentre en collision avec une barre, son ancienne direction est remplacée par une direction qui dépend de la position de l'impact (un peu comme dans un casse brique).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Pour montrer l'utilisation des barres, nous allons ajouter pour ce tutoriel le code nécessaire à la création de 4 barres dans la fonction **recommence** :

La fonction recommence

```
//Recommence la partie
void Jeu::recommence()
{
    Objet o;

    objets.clear();
    Physique::init();
    jeuEnCours = true;

    //Premiere barre
    o.setCouleur(1.0,0.0,0.0);
    o.setPos(WIDTH/2-LARG_BARRE/2,5*HEIGHT/6 - HAUT_BARRE/2);
    o.setTaille(LARG_BARRE,HAUT_BARRE);
    addObjet(o);

    //2eme barre
    o.setCouleur(1.0,1.0,1.0);
    o.setPos(WIDTH/2-LARG_BARRE/2,HEIGHT/6-HAUT_BARRE/2);
    o.setTaille(LARG_BARRE,HAUT_BARRE);
    addObjet(o);

    //3eme barre
    o.setCouleur(0.0,1.0,0.0);
    o.setTaille(HAUT_BARRE,LARG_BARRE);
    o.setPos(WIDTH/6-HAUT_BARRE/2,HEIGHT/2 -LARG_BARRE/2);
    addObjet(o);

    //4eme barre
    o.setCouleur(0.0,1.0,1.0);
    o.setTaille(HAUT_BARRE,LARG_BARRE);
    o.setPos(5*WIDTH/6-HAUT_BARRE/2,HEIGHT/2 -LARG_BARRE/2);
    addObjet(o);
}
```

Comme vous le voyez, ce n'est pas difficile d'ajouter des barres dans le jeu. Nous avons simplement défini en plus la taille souhaitée des barres :

Taille des barres

```
const int LARG_BARRE=120;
const int HAUT_BARRE=25;
```

Bien sûr, qui dit taille de barre, dit taille du quadrillage pour les zones. La taille d'une barre étant plus grande qu'une balle, il faudrait mieux changer le calcul de **QUADCOL** et **QUADLIGNE**.

Voici une solution pour garder le code générique :

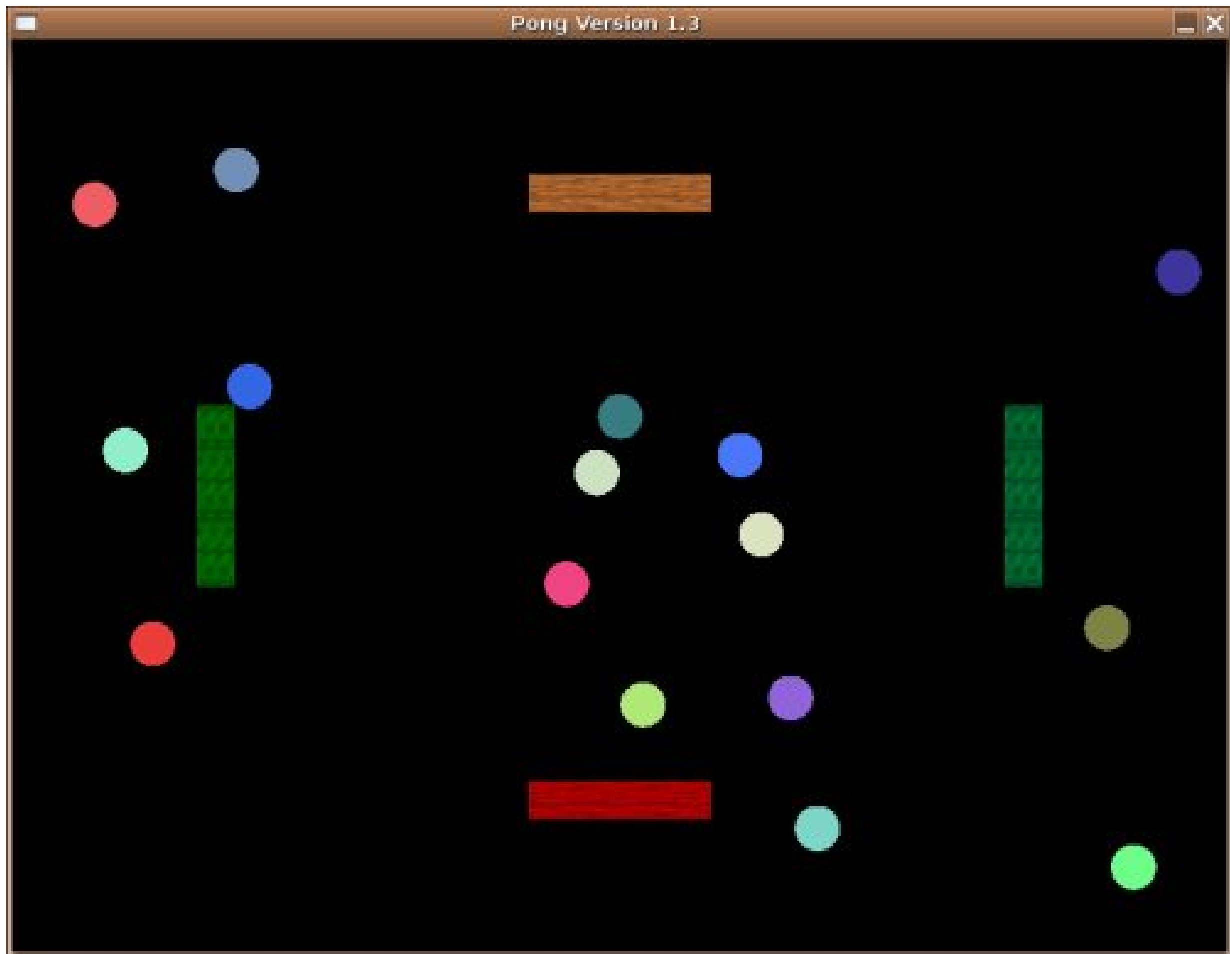
Calcul de QUADCOL et QUADLIGNE

```
const int QUADCOL=WIDTH/ (
            (TAILLE_BALLE>LARG_BARRE)?
                ((TAILLE_BALLE>HAUT_BARRE)?
                    TAILLE_BALLE:
                    HAUT_BARRE):
                ((HAUT_BARRE>LARG_BARRE)?
                    HAUT_BARRE:
                    LARG_BARRE)
           );
const int QUADLIGNE=HEIGHT/ (
            (TAILLE_BALLE>LARG_BARRE)?
                ((TAILLE_BALLE>HAUT_BARRE)?
                    TAILLE_BALLE:
                    HAUT_BARRE):
                ((HAUT_BARRE>LARG_BARRE)?
                    HAUT_BARRE:
                    LARG_BARRE)
           );
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

Certes, l'opérateur ternaire n'est pas l'opérateur favori de beaucoup de programmeurs mais il est pratique ici. En effet, nous avons trois tailles possibles pour ce jeu (et cela demeurera trois pour le reste de cette série) donc nous pouvons nous permettre de faire un calcul sur le maximum des trois tailles possibles : la taille d'une balle, la largeur et la hauteur d'une barre. Ce calcul nous permet de garder sous forme de constante la taille du quadrillage.

Une dernière image montre la présence de quatre barres dans la fenêtre avec vingt balles :



L'introduction des barres

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fearyourself.developpez.com/tutoriel/sdl/pong/partie-4-collisions-ameliorees/

5 - Conclusion

Nous avons vu trois grands concepts dans ce tutoriel. Premièrement, une solution pour réduire considérablement l'impact de la gestion des collisions a été mise en place. En utilisant une solution utilisant une allocation dynamique (utilisant le type **std::vector**), nous avons pu gérer correctement les collisions de 3000 balles...

Ensuite, nous avons amélioré le calcul des déplacements pour le rendre indépendant du taux de rafraîchissement. Cette solution a ajouté une petite fonction pour gérer la mise en pause de jeu.

Enfin, nous avons ajouté un nouveau type d'objet : la barre. Cette barre ne bouge pas comme les balles et n'est pas affectée par les collisions avec les balles. On a pu facilement gérer la nouvelle texture et le comportement des balles lorsqu'elles sont en contact avec la barre.

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.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

6 - Téléchargements

Voici le code source pour ce tutoriel :

- La version avec 3000 balles et allocation statique : (782 Ko) zip.
- La version finale avec l'utilisation du temps dans le mouvement, l'allocation dynamique et l'introduction des barres : (790 Ko) zip.

Voici la version pdf de cet article: (237 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 !

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

7 - Remerciements

J'aimerais remercier loka et khayyam90 pour leur relecture de cet article.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2006 - Jean Christophe Beyler. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.