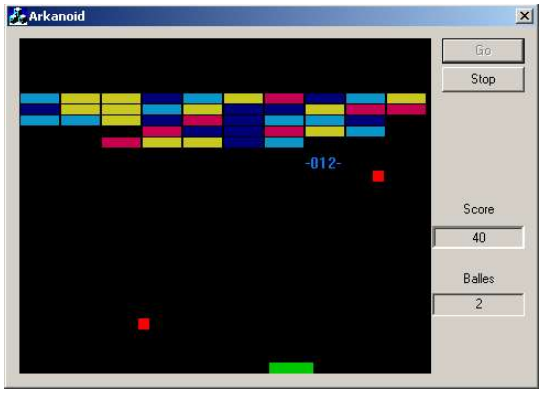
Formation C++
Internet-Entreprise

Tp – Casse briques

Paul Pinault
Clarus-Networks
Paul.pinault@clarus-networks.fr

Objectifs :

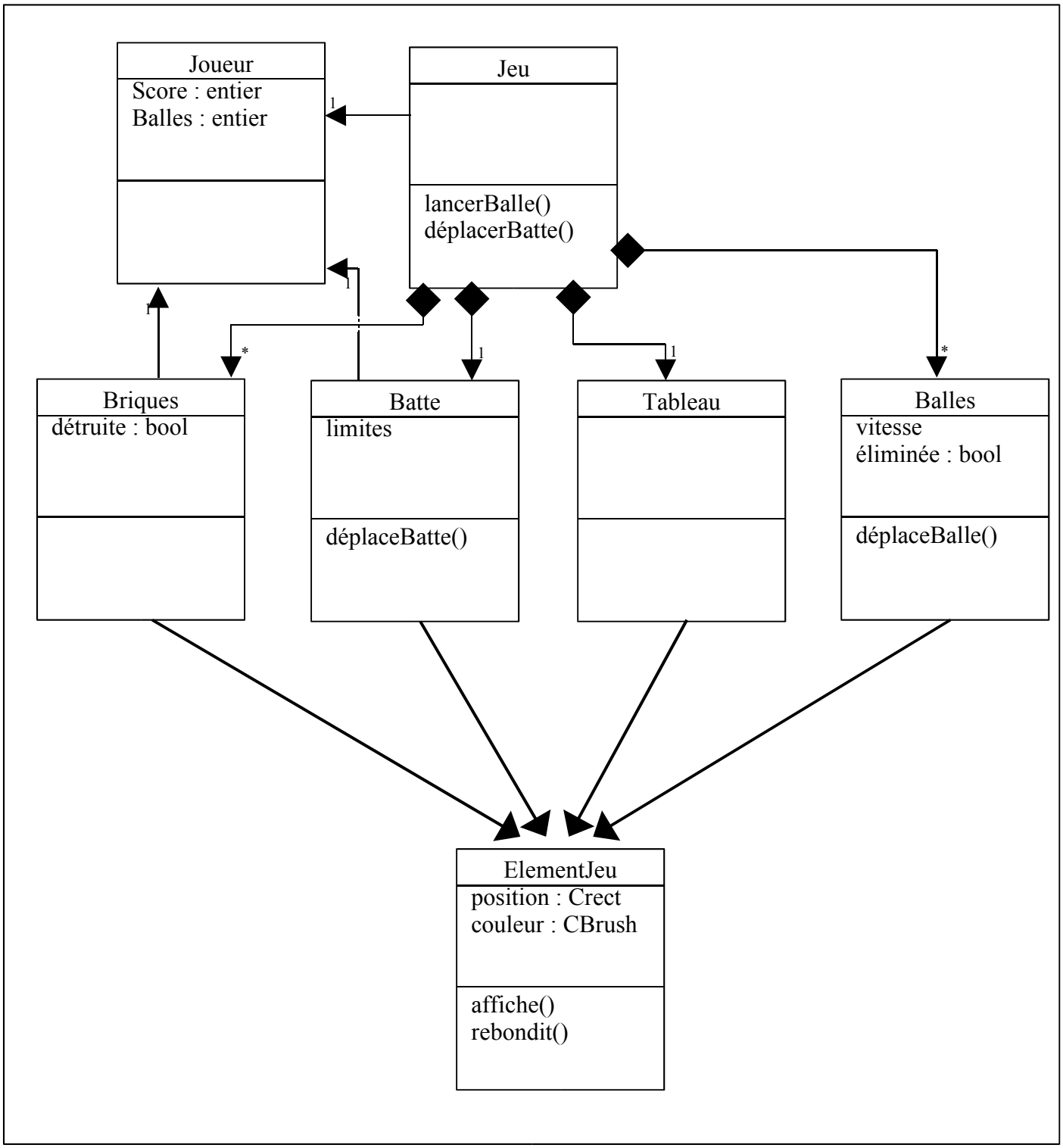
Le but de ce Tp est d'utiliser les concepts objets tels que l'héritage, héritage multiple, les classes virtuelles au travers de la réalisation d'un petit jeu sous Windows : *Arkanoid*.
Le principe du jeu est de détruire les briques présentes sur le tableau à l'aide de balles. Les balles peuvent sortir du jeu si le joueur ne les renvoie pas avec sa batte.



Modélisation d'un casse brique

- Un casse brique est un jeu composé d'un tableau, de briques, d'une batte et de balles. Tous ces éléments sont affichable a l'écran.
- Lorsque la balle touche un élément, il se produit une collision et la balle rebondit. La balle sort du tableau et est éliminée si elle touche le bord inférieur du tableau.
- Un joueur contrôle le jeu en manipulant la batte.
- Une collision entre une balle et une brique détruit la brique et rapporte des points au joueur. Un collision entre une balle et la batte retire des points au joueur.
- Un joueur possède un score et un nombre de balles qu'il peut lancer.
- Les éléments du jeu ont une position dans l'espace du jeu et une couleur.
- Il est possible, a tout moment de lancer de nouvelles balles et de déplacer la batte dans les limites du jeu
- Une balle possède une vitesse et peut se déplacer

1- Réfléchissez au diagramme de classe issu de ces quelques points


Joueur
Score : entier
Balles : entier
Jeu
lancerBalle()
déplacerBatte()
1
1
1
*
1
1
*
Briques
détruite : bool
Batte
limites
déplaceBatte()
Tableau
Balles
vitesse
éliminée : bool
déplaceBalle()
ElementJeu
position : Crect
couleur : CBrush
affiche()
rebondit()

De sorte à voir toutes les étapes de création d'une application dans un environnement Visual C++, nous allons créer pas a pas notre projet.

- Lancez visual C++ puis *File/new/project/MCFAppWizard(exe)*
- Ajoutez un nom de projet : arkanoid
- Vérifiez le chemin du projet
- Clickez ok

- Nous allons choisir une application de type *DialogBased*
- Clickez sur *Next* 2 fois
- Clickez sur *Finish*

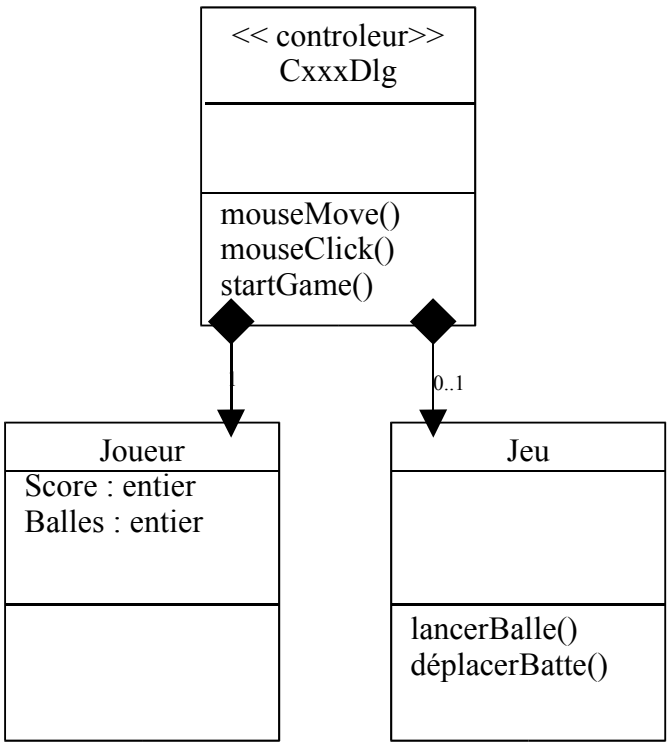
Visual C++ a alors créé un environnement de base pour notre application. Il se compose de 3 classes :

- CaboutDlg : gère l'affichage de la fenêtre « a propos ».
- CxxxApp : objet application lançant les autres objets.
- CxxxDlg : objet graphique représentant l'interface.

Nous allons principalement nous soucier de l'objet CxxxDlg, cet objet est capable de recevoir des événements provenant d'actions de la part de l'utilisateur, en fait il s'agit d'un objet de type **contrôleur** . C'est à dire un objet faisant l'interface entre les objets d'affichage présent sur l'interface et les **objets métiers** que nous allons réaliser.

L'objet contrôleur aura pour rôle la transmission des événements créés par le joueur (physique) à l'objet jeu. Notons que pour ne pas créer plusieurs tableau, on considérera qu'un joueur peut recommencer une nouvelle partie avec ses paramètres actuels : nombre de balles restantes et points obtenus. La fin de la partie à lieu lorsque le joueur n'a plus de balle.

Alors l'objet contrôleur peut être représenté ainsi dans le diagramme de classe :

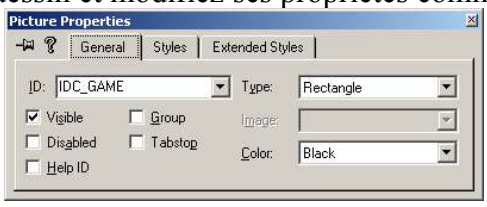


Les mouvements de la souris vont permettre de déplacer la batte, un doubleClick sur le bouton gauche lancera une balle.

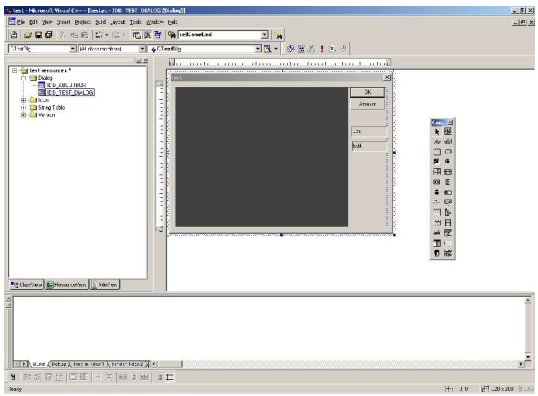
Création de l'interface du jeu

Nous allons maintenant dessiner l'interface de notre jeu de sorte a déjà avoir a disposition tous les éléments graphiques et contrôles. Utilisons l'éditeur de ressources intégré au Visual C++.

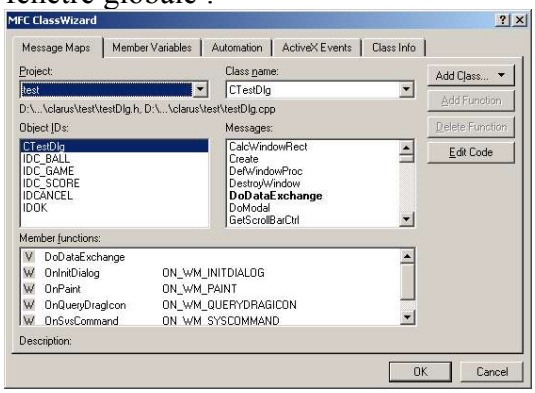
- Supprimez le texte placé au milieu
- Ajouter une zone de dessin et modifiez ses propriétés comme suit :



- Ajoutez deux zones édit box : IDC_SCORE, IDC_BALL de type read_only. Ces zones serviront à indiquer au joueur son score et le nombre de balles qu'il a à sa disposition.



- Ajoutez enfin les contrôles nécessaires à l'interface joueur/jeu en maintenant *CTRL* enfoncé et en double clickant sur la fenêtre globale :



- Ajoutez les contrôles WM_LBUTTONDBLCLK et WM_MOUSEMOVE puis faites ok. Maintenant, deux nouvelles méthodes sont apparues dans le contrôleur : *OnMouseMove*, *OnButtonDblClk*.
- Faites de même pour les objet IDOK et IDCANCEL en ajoutant les contrôles BN_CLICK.

L'interface de notre jeu est maintenant prête, il nous faut écrire les objets métiers que nous interfacerons avec le contrôleur. Une partie des classes vont vous être données pour commencer.

Création des objets métier

La première classe à écrire est ElementJeu , elle est nécessaire à la création des autres classes et ne possède pas de liens vers les autres objets. Voici sa déclaration suivie de son implémentation :

```
class GameElement
{
protected:
        CRect        internalRect;
        CBrush  * internalBrush;
        bool     visible;
public:
        // Un element est un quadrilatere, on utilise la classe CRect existante pour cela
        // et une couleur de remplissage definie par la classe CBrush
        GameElement(CRect myRect, COLORREF myBrush = RGB(0,0,0));

        // Un element est affiché dans un rectangle representant le jeu par la fonction
        // afficheElement
        virtual afficheElement(CDC * theGame);

        // Recuperation des éléments de l'objet
        CRect getRect();

        // Gestion des collisions entre objet : detection de la collision et appel
        // de la fonction virtuelle doCollision. Retourne COL_NONE ou l'indication
        // de localisation de la colision s'il y a lieu
         int performCollision(GameElement * ball);

        // Effectue le traitement necessaire a la gestion de la collision.
         virtual int doCollision(GameElement * ball, int colStatus) = 0;

        // Permet, apres init de modifier la couleur de la brush
         void changeBrush(COLORREF myBrush);

        virtual ~GameElement();
};

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
GameElement::GameElement(CRect myRect, COLORREF myBrush)
{
  internalRect  = myRect;
  internalBrush = new CBrush(myBrush);
  visible = true;
}

GameElement::~GameElement()
{
  delete internalBrush;
}

//////////////////////////////////////////////////////////////////////
// SORTIES A L'ECRAN
//////////////////////////////////////////////////////////////////////
GameElement::afficheElement(CDC * theGame)
{
  if (visible)
    theGame->FillRect(internalRect, internalBrush);
}

//////////////////////////////////////////////////////////////////////
// Acces aux donnees
//////////////////////////////////////////////////////////////////////
CRect GameElement::getRect()
{
        return internalRect;
}

void GameElement::changeBrush(COLORREF myBrush)
{
        CBrush * cb = internalBrush;
        internalBrush = new CBrush(myBrush);
        delete cb;
}

//////////////////////////////////////////////////////////////////////
// COLLISION
//////////////////////////////////////////////////////////////////////
```

```
int GameElement::performCollision(GameElement * ball)
{
  // On determine ici si les deux objets sont en collision
  // Grace aux methodes des rectangle
  CRect intersec;
  if (intersec.IntersectRect(ball->getRect(),internalRect)) {
                // il y a une intersection une collision est donc detectee
                // Cherchons sur quel bord...
                int lstatus = 0;
                int nbord = 0;
                // Test bord bas :
                if ( intersec.bottom == internalRect.bottom ) {
                                nbord++;
                                lstatus = COL_BOTTOM;
                } else {
                // Test bord haut :
                                if ( intersec.top == internalRect.top ) {
                                                nbord++;
                                                lstatus = COL_TOP;
                                }
                }
                // Test bord droit
                if ( intersec.right == internalRect.right ) {
                                nbord++;
                                lstatus = COL_RIGHT;
                } else {
                // Test bord gauche
                                if ( intersec.left == internalRect.left ) {
                                                nbord ++;
                                                lstatus = COL_LEFT;
                                }
                }
                // Cas ou la balle est entierement incluse
               if ( nbord == 0 ) return COL_NONE;

               // Cas des collisions dans les coins
               if ( nbord > 1 ) lstatus = COL_CORNER;

               // Effectue le traitement de collision, appel a une methode
               // virtuelle qui sera definie dans les classes filles...
               return doCollision(ball,lstatus);
    } else {
              // Pas de collision
             return COL_NONE;
    }
}
```

Le système de collision est assez simple, il repose sur le calcul de l'intersection entre deux Elements du jeu. La recherche de collision est une méthode tout a fait commune à tous les objets. Le traitement est par contre lui différent : la collision sur une balle entraîne sont changement de direction, celle d'une brique sa destruction … C'est pourquoi une méthode *doCollision* est appelée pour effectuer le traitement. Cette méthode est virtuelle, elle n'est pas définit dans la classe *GameElement*. Cette classe est donc virtuelle.

Réalisons maintenant les classes *Tableau* et *Balle* et *Game* pour avoir un début de jeu.

```
class Tableau : public GameElement
{
public:
        Tableau(CRect myRect, COLORREF myBrush);
        int doCollision(GameElement *ball, int colStatus);
        virtual ~Tableau();
};

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
Tableau::Tableau(CRect myRect, COLORREF myBrush):GameElement(myRect,myBrush)
{
}

Tableau::~Tableau()
{
}

//////////////////////////////////////////////////////////////////////
// Fonction de gestion des collisions
//////////////////////////////////////////////////////////////////////
int Tableau::doCollision( GameElement * ball, int colStatus )
{
    // la balle rebondit
    return ball->doCollision(this,colStatus);
}
```

Grâce à la création de l'objet *GameElement* l'écriture du tableau est très simple, il suffit d'implémenter la fonction *doCollision* qui n'a pas d'effet sur le *Tableau* et se contente donc d'appeler l'effet sur la balle.

```
class Ball : public GameElement
{
protected:
   // Vecteur vitesse pour la balle
        int  vx;
        int  vy;

public:
        Ball(CRect myRect, COLORREF myBrush);
        int doCollision(GameElement *ball, int colStatus);
        virtual ~Ball(void);
        void moveBall(void);
};

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
Ball::Ball(CRect myRect, COLORREF myBrush):GameElement(myRect,myBrush)
{
        // La direction en x est calculée de façon aléatoire, par contre
        // Vy est fixe pour partir dans la bonne direction.
        vx = (rand() > (RAND_MAX >> 1))?2:-2;
        vy = -2;
}

Ball::~Ball()
{
}
```

```
//////////////////////////////////////////////////////////////////////
// Fonction de gestion des collisions
//////////////////////////////////////////////////////////////////////
// ici un peu special, lorsque la balle entre en collision avec un
// autre objet, on appele la doCollision de Balle avec en param
// l'objet...
int Ball::doCollision( GameElement * ball, int colStatus )
{
   if ( !visible ) return COL_NONE;
   // ... ici pour les effets de la collision
   switch ( colStatus ) {
            case COL_TOP :
            case COL_BOTTOM :     vy = -vy;
                                  break;
            case COL_RIGHT:
            case COL_LEFT:        vx = -vx;
                                  break;
            case COL_CORNER:      vy = -vy;
                                  vx = -vx;
                                  break;
            case COL_LOST:        visible = false;
                                  break;
          }
          return colStatus;
}

//////////////////////////////////////////////////////////////////////
// Deplace la balle
//////////////////////////////////////////////////////////////////////
void Ball::moveBall()
{
   // deplace la balle d'un pixel dans la bonne direction
    if ( visible )  internalRect.OffsetRect(vx,vy);
}
```

Le jeu en version minimaliste est le suivant :

```
class Game
{
protected:
        Tableau     *       gameTab;                // L'objet representant les bords

        Ball        *       gameBall[50];           // Les balles
        int                 numBall;                // Balles mises en jeu au total
        int                 stillBall;                      // Balles restantes en jeu

        CWnd        *       gameWin;                // Objet graphique representant le jeu
        CDC         *       gameDc;
        int                 winW, winH;             // Taille de la fenetre
        CBitmap             gameBitmap;             // Copie Bitmap pour double buffering
        CDC         *       winDc;

        bool                gameEnd;                // Indique que la partie est finie.
        bool                gameWinner;             // Indique que la partie est gagnee.

        CWinThread *        gameThread;
        int                 playerScore;            // memorise le score pour affichage

public:
        int         gameSleep;                      // Temps d'attente

        Game(CWnd * _gameWin, int mySpeed = 10);
        ~Game();
        void paintGame(void);                       // Redessine l'ecran
        void setGameEnd(void);                      // Demande fin du jeu
        bool getGameEnd(void);                      // Interroge etat du jeu
        void startBall(void);                       // Lancer une nouvelle balle
        void actionGame(void);                      // Faire avancer le jeu
};
```

```
//////////////////////////////////////////////////////////////////////
// Processus d'evolution du jeu
//////////////////////////////////////////////////////////////////////
UINT ThreadGame(LPVOID pParam)
{
  Game * mygame = (Game *)(pParam);

  while (!mygame->getGameEnd()) {
              mygame->actionGame();
              mygame->paintGame();
             Sleep(mygame->gameSleep);
  }
  return 0;
}


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
Game::Game(CWnd * _gameWin,  int mySpeed )
{

  // Extraction des informations provenant de l'objet graphique
  gameWin  = _gameWin;
  winDc    = gameWin->GetDC();
  CRect   rtab;
  gameWin->GetClientRect(&rtab);
  // Creation du BitMap interne
  winW = rtab.Width();
  winH = rtab.Height();
  gameBitmap.CreateCompatibleBitmap(winDc,winW,winH);
  gameDc = new CDC();
  gameDc->CreateCompatibleDC(winDc);
  gameDc->SelectObject(&gameBitmap);

  // Init des variables interne a l'objet
  gameEnd    = false;
  gameWinner = false;
  numBall    = 0;
  stillBall  = 0;
  gameSleep  = mySpeed;

  // Creation des objets metier
  gameTab    = new Tableau(rtab, RGB(0,0,0));

  // Lancement du thread associé au jeu
  gameThread = AfxBeginThread(ThreadGame,this,THREAD_PRIORITY_TIME_CRITICAL);
}

Game::~Game()
{
   // Il vaut mieux que le thread soit mort !
            gameEnd = true;
            Sleep(gameSleep*4);

            delete gameTab;
            for ( int i = 0 ; i < numBall   ; i++ ) delete gameBall[i];
}


//////////////////////////////////////////////////////////////////
// Parametres
//////////////////////////////////////////////////////////////////
bool Game::getGameEnd()
{
            return gameEnd;
}
void Game::setGameEnd()
{
            gameEnd = true;
}
//////////////////////////////////////////////////////////////////
// Affichage
```

```
//////////////////////////////////////////////////////////////
void Game::paintGame()
{
   // Netoyage de l'ecran
   gameTab->afficheElement(gameDc);

   if ( !gameEnd ) {

            // Affiche les elements du jeu
            for ( int  i = 0 ; i < numBall   ; i++ ) gameBall[i]->afficheElement(gameDc);

  } else {

            // Affiche Game Over/Win
            gameDc->SetTextColor(RGB(255,0,0));
            gameDc->SetBkColor(RGB(0,0,0));
            if ( !gameWinner ) {
                     gameDc->TextOut(150,150,"Game Over !");
            } else gameDc->TextOut(150,150,"Game Win !");
   }

   // Met a jour la zone graphique reelle
  winDc->BitBlt(0,0,winW,winH,gameDc,0,0,SRCCOPY);
  gameWin->UpdateWindow();
}

//////////////////////////////////////////////////////////////
// Lancement de la balle
//////////////////////////////////////////////////////////////
void Game::startBall()
{
        gameBall[numBall] = new Ball(new CRect( 10,10,20,20 ),
                                     RGB(255,0,0));
        numBall++;
       stillBall++;
}

//////////////////////////////////////////////////////////////
// Evolution de la balle et du jeu
//////////////////////////////////////////////////////////////
void Game::actionGame()
{
   // pour chaque balle, on effectue son mouvement et on teste
   // les collisions avec les autres objets du jeu...
   for ( int i = 0 ; i < numBall ; i++ ) {
            gameBall[i]->moveBall();

            if ( gameTab->performCollision(gameBall[i]) == COL_LOST ) {
                if ( --stillBall == 0 ) {
                    gameEnd    = true;
                    gameWinner = false;
                }
            }
   }
}
```

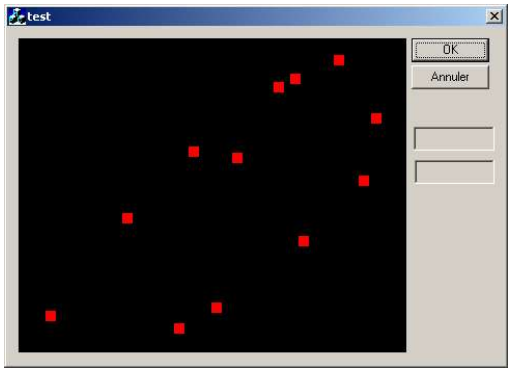
La classe *Game* utilise un Processus indépendant pour faire évoluer le jeu, elle le lance dès que les éléments du jeu son créés. Il sera terminé lorsque le jeu sera marqué finit.
Lors d'une évolution du jeu, les balles sont déplacées et leur collision avec les autres objets du jeu est testée. L'affichage redemande simplement l'affichage de chaque élément ; si le jeu est fini, affiche un message de fin gagnante ou perdante selon le cas. Le jeu est perdu si la dernière balle en jeu est perdue.

Intégration avec le contrôleur

Ajoutons au contrôleur un objet *Game* *myGame* qui sera initialisé dans la méthode *OnInitDialog* à NULL. La méthode *OnOk* est alors utilisée pour démarrer le jeu. La methode *OnLButtonDblClk* est utilisée pour lancer une nouvelle balle :

```
BOOL CxxxxDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        [...]

        // TODO: Add extra initialization here
        myGame = NULL;
        return TRUE;  // return TRUE  unless you set the focus to a control
}

void CxxxxDlg::OnOK()
{
        // TODO: Add extra validation here
        if ( myGame ) delete myGame;
        myGame = new Game(GetDlgItem(IDC_GAME),10);
}

void CxxxxDlg::OnLButtonDblClk(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default
        if (myGame) myGame->startBall();
        CDialog::OnLButtonDblClk(nFlags, point);
}
```

Vous pouvez maintenant compiler et lancer l'application…



Exercice :

2- **Créez une classe joueur comptabilisant les points et indiquant le nombre de balles disponibles. Modifier la classe *Game* pour restreindre le lancement des balles a la disponibilité des balles.**
3- **Modifiez la classe *Tableau* pour gérer la sortie des balles lorsqu'elles tapent au bas de celui-ci.**

Indication : pour modifier le texte dans les cases score et balles, utilisez la commande ci-dessous :

```
CWND * ScoreWin = GetDlgItem(IDC_SCORE) ;
ScoreWin->setWindowText(« chaine_de_char ») ;
```

Ne faites aucun traitement dans le contrôleur.

Voici la classe Joueur :

```
class Joueur
{
protected:
	CWnd	*	scoreWin;		// Objet champ texte du score
	CWnd	*	balleWin;		// Objet champ texte du nombre de balles
	int		score;			// Score du joueur
	int		balle;			// Nombre de balles restantes pour le joueur
public:
	Joueur(CWnd * _scoreWin, CWnd * _balleWin);
	virtual ~Joueur();
	 void donnerPoint(int points);
	void resetScore(void);
	 bool haveBall(void);
	void updateTxt(void);
	void addBall(int num);
	int  getScore(void);
};
////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////
Joueur::Joueur(CWnd * _scoreWin, CWnd * _balleWin)
{
  scoreWin = _scoreWin;   balleWin = _balleWin;
  score    = 0;   balle    = 5;
}
Joueur::~Joueur(){ }
////////////////////////////////////////////////////////////////////
// MISE A JOUR SCORE
////////////////////////////////////////////////////////////////////
void Joueur::donnerPoint(int points)
{  score += points;		updateTxt();  }

void Joueur::resetScore()
{ score = 0; balle = 5; updateTxt(); }

int Joueur::getScore()
{ return score; }
////////////////////////////////////////////////////////////////////
// Affichage des informations du joueur
////////////////////////////////////////////////////////////////////
void Joueur::updateTxt()
{
	// Mise a jour score
	char strScore[20];
	sprintf(strScore,"%d",score);
	 scoreWin->SetWindowText(strScore);
	scoreWin->UpdateWindow();
	sprintf(strScore,"%d",balle);
	 balleWin->SetWindowText(strScore);
	balleWin->UpdateWindow();
}
////////////////////////////////////////////////////////////////////
// MISE A JOUR BALLES
////////////////////////////////////////////////////////////////////
// Le joueur utilise une balle
bool Joueur::haveBall()
{
	if ( balle )
	{  balle--; updateTxt(); return true; }
	return false;
}

// Le joueur gagne des balles
void Joueur::addBall(int num)
{
	balle += num;updateTxt();
}
```

Les modifications dans le contrôleur :

```
BOOL CTestDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
	[...]
	myGame = NULL;
	myPlayer = new Joueur(GetDlgItem(IDC_SCORE),GetDlgItem(IDC_BALL));
	myPlayer->updateTxt();
	return TRUE;  // return TRUE  unless you set the focus to a control
}
void CTestDlg::OnOK()
{
	 if ( myGame ) delete myGame;
	myGame = new Game(GetDlgItem(IDC_GAME),myPlayer,10);
}
```

Dans la classe *Game*

```
class Game
{
protected:
	Tableau		*	gameTab;		// L'objet representant les bords
	Joueur		*	gamePlayer;		// Le joueur
	[...]
public:
	int		gameSleep;		// Temps d'attente
	Game(CWnd * _gameWin, Joueur * myPlayer, int mySpeed = 10);
	~Game();
	[...]
} ;

Game::Game(CWnd * _gameWin, Joueur * myPlayer, int mySpeed )
{
  // Extraction des informations provenant de l'objet graphique
  [...]
  // Init des variables interne a l'objet
  gamePlayer = myPlayer;
  [...]
}
//////////////////////////////////////////////////////////////////
// Lancement de la balle
//////////////////////////////////////////////////////////////////
void Game::startBall()
{
	if ( gamePlayer->haveBall() ) {
		gameBall[numBall] = new Ball(new CRect(10,10,20,20 ), RGB(255,0,0));
		numBall++;
		stillBall++;
	}
}
```

Enfin les modification de tableau pour le test de sortie de la balle :

```
//////////////////////////////////////////////////////////////////////
// Fonction de gestion des collisions
//////////////////////////////////////////////////////////////////////
int Tableau::doCollision( GameElement * ball, int colStatus )
{
   if ( colStatus == COL_BOTTOM ) colStatus = COL_LOST;
   return ball->doCollision(this,colStatus);
}
```

4 – Créez la classe Batte se déplaçant à l'aide de la souris. Attention à ce qu'elle ne sorte pas de la zone de jeu. La méthode *OnMouseMove* est appelée à chaque mouvement de souris avec en paramètre la coordonnée du pointeur. Une collision de balle avec la batte retire 10 points au joueur.

La classe Batte

```
class Batte : public GameElement
```

```
{
protected:
        int min_x, max_x;           // limites de deplacement de la batte
        Joueur * internalPlayer;

public:
        Batte(CRect myRect, COLORREF myBrush, Joueur * myPlayer);
        virtual ~Batte();
        void setMinMaxX(int minx, int maxx);
        void moveBatte(int deplacement);
         int doCollision(GameElement *ball, int colStatus);
};
////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////
Batte::Batte(CRect myRect, COLORREF myBrush, Joueur * myPlayer):GameElement(myRect,myBrush)
{
  min_x = 0; max_x = 200;
  internalPlayer = myPlayer;
}
Batte::~Batte(){ }
////////////////////////////////////////////////////////////////////
// Deplacements de la batte ( horizontal )
////////////////////////////////////////////////////////////////////
void Batte::moveBatte(int deplacement)
{
   deplacement = deplacement - internalRect.right;
   if (    ( (internalRect.left  + deplacement) >= min_x )
                && ( (internalRect.right + deplacement) <= max_x ) ) {
      internalRect.OffsetRect(deplacement,0);
        }
}
void Batte::setMinMaxX(int minx, int maxx)
{
  min_x = minx; max_x = maxx;
}
////////////////////////////////////////////////////////////////////
// Fonction de gestion des collisions
////////////////////////////////////////////////////////////////////
int Batte::doCollision( GameElement * ball, int colStatus )
{
    int ret = ball->doCollision(this,colStatus);
    if ( ret != COL_NONE ) internalPlayer->donnerPoint(-10);
    return ret;
}
```

Les modifications dans Game :

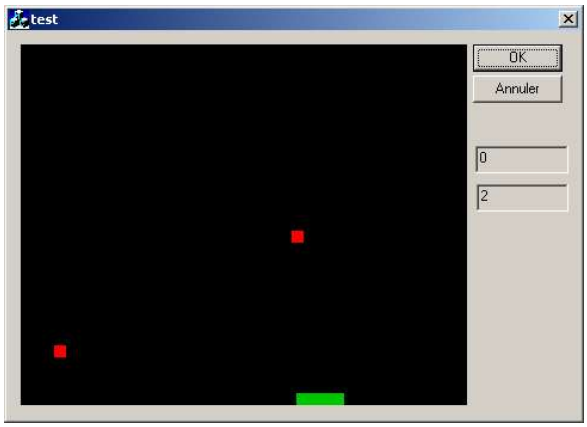
```
class Game
{
protected:
        Tableau         *       gameTab;        // L'objet representant les bords
        Joueur          *       gamePlayer;     // Le joueur
        Batte           *       gameBatte;      // La batte
        Ball            *       gameBall[50];   // Les balles
        [...]
public:
        [...]
        void moveBatte(CPoint p);               // Deplace la batte
};
```

```
Game::Game(CWnd * _gameWin, Joueur * myPlayer, int mySpeed )
{
  [...]
  // Creation des objets metier
  gameTab    = new Tableau(rtab, RGB(0,0,0));
  gameBatte  = new Batte(CRect( (rtab.Width()/2)-20,
                                (rtab.bottom -10),
                                (rtab.Width()/2)+20,
                                (rtab.bottom) ),
                         RGB(0,200,0),
                         gamePlayer );
   gameBatte->setMinMaxX(rtab.left,rtab.right);
   // Lancement du thread associé au jeu
   gameThread = AfxBeginThread(ThreadGame,this,THREAD_PRIORITY_TIME_CRITICAL);
}

Game::~Game()
{
        [..]
        delete gameTab;
        delete gameBatte;
        for ( int i = 0 ; i < numBall   ; i++ ) delete gameBall[i];
}

////////////////////////////////////////////////////////////
// Déplacement de la batte
////////////////////////////////////////////////////////////
void Game::moveBatte(CPoint p)
{
        gameBatte->moveBatte(p.x);
}

////////////////////////////////////////////////////////////
// Affichage
////////////////////////////////////////////////////////////
void Game::paintGame()
{
   // Netoyage de l'ecran
   gameTab->afficheElement(gameDc);
   if ( !gameEnd ) {
        // Affiche les elements du jeu
        for ( int i = 0 ; i < numBall   ; i++ ) gameBall[i]->afficheElement(gameDc);
        gameBatte->afficheElement(gameDc);
   } else {
   [...]
}
////////////////////////////////////////////////////////////
// Evolution de la balle et du jeu
////////////////////////////////////////////////////////////
void Game::actionGame()
{
   for ( int i = 0 ; i < numBall ; i++ ) {
        gameBall[i]->moveBall();
        gameBatte->performCollision(gameBall[i]);
  [...]
}
```

```
////////////////////////////////////////////////////////////
// Lancement de la balle
////////////////////////////////////////////////////////////
void Game::startBall()
{
    if ( gamePlayer->haveBall() ) {
      gameBall[numBall] = new Ball(new CRect( gameBatte->getRect().left
                                              + (gameBatte->getRect().Width()/2) -5,
                                              gameBatte->getRect().top-10,
                                              gameBatte->getRect().left
                                              + (gameBatte->getRect().Width()/2) +5,
                                              gameBatte->getRect().top ),
                                   RGB(255,0,0));
    [...]
}
```

Modifications dans le contrôleur :

```
void CTestDlg::OnMouseMove(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default
        if ( myGame ) myGame->moveBatte(point);
        CDialog::OnMouseMove(nFlags, point);
}
```



5 – Créez une classe Brique, les briques se cassent lors d'une collision avec une balle et donnent 20 points au joueur. Le niveau sera créé comme suit :

```
////////////////////////////////////////////////////////////
// Creation niveau
////////////////////////////////////////////////////////////
void Game::createLevel()
{  int    ran;   CRect trect;   CRect rtab;   gameWin->GetClientRect(&rtab);
   numBrique = 0;
   for ( int y = 0 ; y < 5  ; y++) {
            for ( int x = 0 ; x < 10 ; x++) {
                trect.SetRect((rtab.Width()/10)*x+1, (y+5)*10, (rtab.Width()/10)*(x+1)-1, (y+6)*10-1 );
                ran = rand() % 10;
                switch (ran) {
                    case 0:    case 1:   case 2:    case 3:    case 4:
                    case 5:    case 6:   case 7:    case 8:    case 9:
                                    gameBrique[numBrique] = new Brique(trect,gamePlayer);
                                break;
                  }
                numBrique++;
            }
   }
}
```

Classe Brique

```
class Brique : public GameElement
{
protected:
        Joueur   *          internalPlayer;
public:
        Brique(CRect myRect, Joueur * myPlayer);
        virtual ~Brique();
         int doCollision( GameElement * ball, int colStatus );
        bool getStatus(void);
};
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
Brique::Brique(CRect myRect, Joueur * myPlayer):GameElement(myRect)
{
  GameElement::changeBrush(RGB(200,200,30));
  internalPlayer = myPlayer;
}

Brique::~Brique(){}
//////////////////////////////////////////////////////////////////////
// Fonction de gestion des collisions
//////////////////////////////////////////////////////////////////////
int Brique::doCollision( GameElement * ball, int colStatus )
{
        if ( !visible ) return COL_NONE;

        // La brique est notée détruite
        visible = false;

        int ret = ball->doCollision(this,colStatus);
        if ( ret != COL_NONE )
          internalPlayer->donnerPoint(20);
         return ret;
}
//////////////////////////////////////////////////////////////////////
// ACCES A l'etat
//////////////////////////////////////////////////////////////////////
// revoie true si la brique n'est plus visible (detruite)
bool Brique::getStatus()
{
        return ((visible)?false:true);
}
```

Modifications de Game

```
class Game
{
protected:
        Ball        *       gameBall[50];           // Les balles
        int                 numBall;                // Balles mises en jeu au total
        int                 stillBall;              // Balles restantes en jeu

        Brique      *       gameBrique[50];         //  Les briques
        int                 numBrique;
        [...]
protected:
         void               createLevel(void);
        [...]
};
```

```
////////////////////////////////////////////////////////////
// Evolution de la balle et du jeu
////////////////////////////////////////////////////////////
void Game::actionGame()
{
        // pour chaque balle, on effectue son mouvement et on teste
        // les collisions avec les autres objets du jeu...
        for ( int i = 0 ; i < numBall ; i++ ) {
                [...]
                for ( int k = 0 ; k < numBrique ; k++ )
                    gameBrique[k]->performCollision(gameBall[i]);
        }

        // Teste la fin gagnante
        bool fin = true;
        for ( i = 0 ; i < numBrique ; i++ )
                if ( !gameBrique[i]->getStatus() ) fin = false;
        if ( fin ) {
                gameEnd = true;
                gameWinner = true;
        }
}

////////////////////////////////////////////////////////////
// Affichage
////////////////////////////////////////////////////////////
void Game::paintGame()
{
   // Netoyage de l'ecran
   gameTab->afficheElement(gameDc);

   if ( !gameEnd ) {

        // Affiche les elements du jeu
        for ( int i = 0 ; i < numBall   ; i++ ) gameBall[i]->afficheElement(gameDc);
        for (     i = 0 ; i < numBrique ; i++ ) gameBrique[i]->afficheElement(gameDc);
        gameBatte->afficheElement(gameDc);
   [...]
}
```

6 – Créez une nouvelle classe : les briques dures qui demandent deux coups pour être détruites. La brique change de couleur après le premier coup. Pour ceux en avance, vous pouvez créer des briques molles (qui n'entraînent pas de rebondissement de la balle) .

7 – A l'aide le la classe *Sound* ci-dessous et par héritage multiple ajoutez des bruitages lors des collisions entre brique et balle et entre batte et balle.

```
class Sound
{
protected:
        CString         * soundName;

public:
        Sound(char * sndName = NULL);
        virtual ~Sound();
        playSound();
        setSound(char * sndName);
};
```

Rq : vous ajouterez winmm.lib à la liste des librairies lors le la compilation.

```
Sound::Sound(char * sndName) {
        if (sndName)     soundName = new CString(sndName);
        else   soundName = NULL;
}
Sound::setSound(char * sndName) {
        if (sndName)    soundName = new CString(sndName);
        else   soundName = NULL;
}
Sound::~Sound() {
        if (soundName) delete soundName;
}
Sound::playSound()
{
   if(soundName)
      PlaySound(soundName->GetBuffer(0),::AfxGetInstanceHandle(),
                SND_RESOURCE | SND_ASYNC );
}
```

La classe BriqueDure :

```
class BriqueDure : public Briqu
{
protected:
        int  touche;

public:
        BriqueDure(CRect myRect, Joueur * myPlayer);
        virtual ~BriqueDure();
         int doCollision( GameElement * ball, int colStatus );
};

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
BriqueDure::BriqueDure(CRect myRect, Joueur * myPlayer):Brique(myRect,myPlayer)
{
        GameElement::changeBrush(RGB(0,0,120));
        touche = 0;
}

BriqueDure::~BriqueDure(){ }

//////////////////////////////////////////////////////////////////////
// Fonction de gestion des collisions
//////////////////////////////////////////////////////////////////////
int BriqueDure::doCollision( GameElement * ball, int colStatus )
{
        int ret;
        ret = Brique::doCollision(ball,colStatus);
        if ( ret != COL_NONE ) {
          // Finalement la brique n'est détruite que si elle est touchée deux fois
          if (++touche < 2) {
                        visible = true;
                        GameElement::changeBrush(RGB(0,0,255));
          }
        }
        return ret;
}
```

Modification dans Game :

```
//////////////////////////////////////////////////////////////
// Creation niveau
//////////////////////////////////////////////////////////////
void Game::createLevel()
{
  int    ran;   CRect trect;   CRect rtab;   gameWin->GetClientRect(&rtab);
  numBrique = 0;
  for ( int y = 0 ; y < 5  ; y++) {
```

```
                for ( int x = 0 ; x < 10 ; x++) {
                    trect.SetRect((rtab.Width()/10)*x+1,  (y+5)*10,(rtab.Width()/10)*(x+1)-1,(y+6)*10-1 );
                    ran = rand() % 10;
                    switch (ran) {
                        case 0:   case 1:   case 2:   case 3:
                                    gameBrique[numBrique] = new BriqueDure(trect,gamePlayer);
                                    break;
                        case 4:   case 5:   case 6:   case 7:   case 8:   case 9:
                                    gameBrique[numBrique] = new Brique(trect,gamePlayer);
                                    break;
                    }
                   numBrique++;
        } } }
```

Ajout du son pour la Batte :

```
class Batte : public GameElement, public Sound
{
protected:
        [...]
};

Batte::Batte(CRect myRect, COLORREF myBrush, Joueur * myPlayer):
GameElement(myRect,myBrush),Sound()
{
  min_x = 0; max_x = 200;
  internalPlayer = myPlayer;
  setSound("IDR_BATTE");
}
int Batte::doCollision( GameElement * ball, int colStatus )
{
        int ret = ball->doCollision(this,colStatus);
        if ( ret != COL_NONE ) {
                internalPlayer->donnerPoint(-10);
                playSound();
        }
        return ret;
}
```

Les sons sont inclus dans les ressources : « ressources » / « import – wave ». Le nom de la ressources doit être modifié pour devenir « ''IDR_BATTE'' ».