# Programmation Java sous Android

Malgré le procès qu'Oracle, l'actuel détenteur et promoteur du langage de programmation Java, serait en train de perdre face à Google, Java restera certainement encore pour quelques temps le langage de programmation de prédilection des appareils mobiles tournant sous le système d'exploitation Android. C'est après la perte d'un procès semblable face à l'entreprise Sun, rachetée depuis par Oracle, que Microsoft avait préféré se détourner de Java pour créer en toute impunité un clone maison, C#. Laissons les affaires de gros sous et les deux Larry (Ellison et Page) s'étriper, et intéressons-nous plutôt à la programmation objet, et plus spécifiquement Java, à laquelle les programmeurs d'application mobile sous Android doivent s'atteler.

Les usages des smartphones et autres tablettes se veulent différents de ceux des ordinateurs : téléphone, SMS, photos et musique, géolocalisation, GPS… Par ailleurs, un CPU moins puissant et une mémoire plus petite obligent à des optimisations de code qui ne s'accordent pas parfaitement aux principes de la programmation objet, principes présentés et âprement défendus dans cet ouvrage.

Nous avons déjà eu l'occasion de l'argumenter ; il doit vous être acquis, à ce stade de la lecture, que la programmation objet se préoccupe autrement plus du temps développeur que du temps CPU. Modularité, encapsulation, héritage, polymorphisme, autant de recettes de programmation qui coûtent assez cher en temps processeur mais qui facilitent la vie du développeur et garantissent une certaine stabilité au cours de ce développement. Or, à mon grand désarroi, il n'est pas rare de trouver dans les ouvrages de développement d'applications sous Android des recommandations telles que « évitez les méthodes `getters` et `setters` », « ne recourez pas aux interfaces ou autres polymorphismes qui obligent le processeur à retrouver la bonne méthode à exécuter durant le run time, en s'affranchissant ainsi du temps que le compilateur permettrait de faire gagner ».

Je ne pense pas vous surprendre davantage en vous annonçant tout de go vouloir camper sur mes positions et continuer coûte que coûte à promouvoir et défendre la charte de la bonne programmation objet en pariant sur l'amélioration des performances des appareils mobiles à venir (pourquoi la divine loi de Moore ne s'appliquerait-elle pas à eux également ?). On ne se refait pas ! Le dur labeur du développeur m'importe plus que la rapidité de sa géolocalisation.

Le Java à utiliser sur Android est aussi quelque peu différent que le Java d'origine, celui que j'ai exploité dans l'essentiel de cet ouvrage (d'où le procès contre Google). Pour des raisons de performance et de compatibilité avec le système d'exploitation Android, ce n'est pas la Java Virtual Machine d'origine qui exécute les codes Java mais une autre machine virtuelle dénommée Dalvik. Cette dernière fait un large usage des registres plutôt que de la pile dans l'implémentation des instructions bytecodes (le résultat de la compilation d'un code Java). Les processeurs ARM présents dans la grande majorité des smartphones et tablettes sont de type RISC et donc conçus pour faire un usage intensif des registres dans l'exécution des instructions élémentaires.

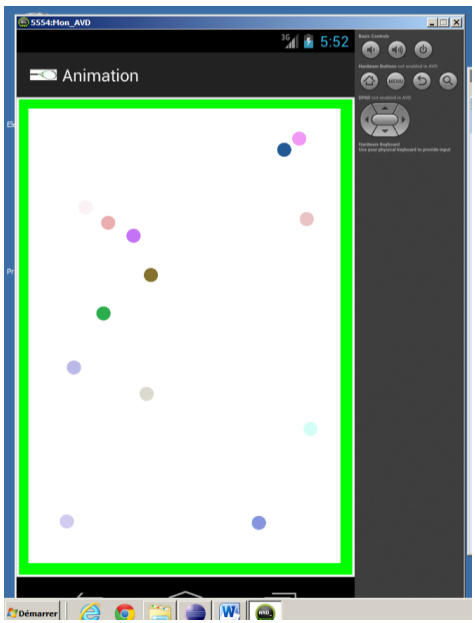
À la différence de l'exécution habituelle des programmes Java, chaque application développée sous Android requiert pour s'exécuter sa propre instance de la machine Dalvik. Cela permet une plus grande flexibilité et robustesse dans l'interruption de ces mêmes applications, interruption décidée par le système Android lui-même (et personne d'autre) lorsqu'il se trouve en manque de ressource. Aussi, de nombreuses librairies Java ont été réécrites de manière à mieux s'interfacer avec le système d'exploitation Android, rompant quelque peu avec la philosophie initiale de Java de se programmer d'une seule et même manière quelle que soit la plate-forme censée exécuter votre code. Ces nouvelles librairies Java sont la raison principale des différences entre les codes présentés ci-dessous et ceux du chapitre 20.

Le but de ce chapitre additionnel n'est en aucune sorte de vous initier à la programmation sous Android. Le sujet est bien trop conséquent et exige un apprentissage dédié. Ces quelques pages ne pourraient nullement y suffire et les librairies regorgent d'ouvrages endossant ce rôle mieux que je ne pourrais le faire (je ne programme sous Android qu'à titre de pédagogie de la programmation objet). Néanmoins, il m'a paru intéressant dans un premier temps de permettre aux lecteurs de l'ouvrage de faire tourner sous Android, sur leur téléphone ou tablette, deux des applications graphiques présentées dans le chapitre 20 : la petite animation et le tennis. Au fur et à mesure de la présentation des codes, je me permettrai quelques explications m'obligeant à quelques digressions dans l'univers du développement Android, mais une formation additionnelle au développement sous Android s'avère

indispensable pour comprendre ces deux codes et les exécuter en l'état sur votre téléphone ou tablette).

## Animation balles colorées

La petite animation représentée ci-dessous (et s'exécutant dans un émulateur Android) comprend une liste de balles colorées qui s'entrechoquent et rebondissent sur les parois vertes.




Quatre classes sont nécessaires pour la réalisation de cette animation. Les parties visibles des applications Android s'appellent des classes `Activity` et se programment comme suit (en héritant de la classe `Activity`).

**Classes Activity**

```
public class MainActivity extends Activity implements
OnTouchListener {
   DrawView drawView;
   protected static final int GUIUPDATEIDENTIFIER = 0;

   Handler myGUIUpdateHandler = new Handler() {

      @Override
      public void handleMessage(Message msg){
         switch (msg.what){
```

```
            case MainActivity.GUIUPDATEIDENTIFIER:
               drawView.update();
               drawView.invalidate();
               super.handleMessage(msg);
               break;
            }

         }
      };

      @Override
      public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         drawView = new DrawView(this);
         drawView.setOnTouchListener(this);
         drawView.setBackgroundColor(Color.WHITE);
         setContentView(drawView);
         new Thread (new RefreshRunner()).start();
      }

      class RefreshRunner implements Runnable {
         // @Override
         public void run() {
            while (!Thread.currentThread().isInterrupted()) {
               /* envoie un Message au Handler qui va appeler
la méthode invalidate() sur le DrawView */
               Message message = new Message();
               message.what =
MainActivity.GUIUPDATEIDENTIFIER;

MainActivity.this.myGUIUpdateHandler.sendMessage(message);

               try {
                  Thread.sleep(2);
               } catch (InterruptedException e) {
                  Thread.currentThread().interrupt();
               }
            }
         }
      }

      public boolean onTouch(View v, MotionEvent e) {
         switch (e.getAction()) {
         case MotionEvent.ACTION_DOWN:
            int x = (int)e.getRawX() - 70;
            int y = (int)e.getRawY() - 130;
            drawView.lesBalles.add(new Balle(x,y,drawView));
         }
         return true;
      }

}
```

Le graphisme de cette application s'affiche dans un `Drawview` que nous présenterons plus bas (et qui hérite de la classe `View`, comme tout ce qui s'affiche sur Android). Le démarrage de l'`activity` déclenche d'office la méthode `onCreate` qui, ici, se contente de créer un objet `Drawview`, de le peindre en blanc, et de le rendre sensible au « toucher de souris » (à chaque clic, une nouvelle balle est créée).

Nous retrouvons le multithreading du code d'origine. Un thread est créé et démarré sous la forme de la classe `RefreshRunner`. Plusieurs options sont possibles pour l'implémentation du multithreading sous Android. J'ai choisi ici, par facilité, de faire appel à la classe `Handler` dont le rôle est de faire circuler des messages entre threads. Les applications Android fonctionnent en grande partie par des envois de messages entre elles afin de se solliciter mutuellement. Dans ce code, le thread additionnel n'a d'autre fonction que de freiner l'activité un certain temps (présence du `sleep`), puis de déclencher le réaffichage en envoyant un message au thread principal, celui en charge de l'affichage. À la réception de ce message, le thread principal met à jour l'application et appelle la méthode `invalidate`. Celle-ci se contente de redessiner toutes les balles et les parois. La méthode `onTouch` provenant de l'interface `OnTouchListener` décrit ce qui se passe lors d'un clic souris : le rajout d'une balle à l'endroit du clic.

**Classe DrawView**

```
public class DrawView extends View   {
   public Paint paint;
   public ArrayList<Balle> lesBalles;
   public Parois[] lesParois;
   public int canvasW = 500;
   public int canvasH = 600;


   public DrawView(Context context) {
      super(context);
      paint = new Paint();
      lesParois = new Parois[4];
      //ourHolder = getHolder();
      lesBalles = new ArrayList<Balle>();
      lesBalles.add(new Balle(50,50,this));
      lesParois[0] = new Parois(5, 5, 25, canvasH,this);
      lesParois[1] = new Parois(5, 5, canvasW, 25,this);
      lesParois[2] = new Parois(5, canvasH, canvasW,
canvasH + 25,this);
      lesParois[3] = new Parois(canvasW, 5, canvasW+25,
canvasH + 25,this);
   }

   @Override
```

```
   public void onSizeChanged (int w, int h, int oldw, int
oldh){
      super.onSizeChanged(w, h, oldw, oldh);
      canvasW = w - 30;
      canvasH = h - 30;
      lesParois[0].changeDimension(5,5,25,canvasH);
      lesParois[1].changeDimension(5, 5, canvasW, 25);
      lesParois[2].changeDimension(5, canvasH, canvasW,
canvasH + 25);
      lesParois[3].changeDimension(canvasW, 5, canvasW+25,
canvasH + 25);
   }

   public void onDraw(Canvas canvas) {
      for (Balle b : lesBalles) {
         b.dessineToi(canvas);
      }
      for(Parois p : lesParois) {
         p.dessineToi(canvas);
      }
   }

   public void update() {
      for (Balle b : lesBalles) {
         b.bouge(lesParois, lesBalles);
      }
   }

}
```

La classe `DrawView` est en charge de l'initialisation et de l'affichage des balles et des parois. La méthode `invalidate` appelle implicitement la méthode `onDraw(Canvas)` qui, à partir de l'argument `Canvas` qu'elle reçoit, dessinera les formes primitives des balles et des parois.

Rien de bien particulier pour les deux dernières classes (par rapport au code déjà décrit dans le chapitre 20), `Balle` et `Parois`, qui font toutes deux usage de la classe `RectF` pour la prise en charge des intersections et se dessine à l'aide des deux classes `DrawView` (plus particulièrement l'objet `paint` associé à cette classe ») et `Canvas`.

**Classes Balle et Parois**

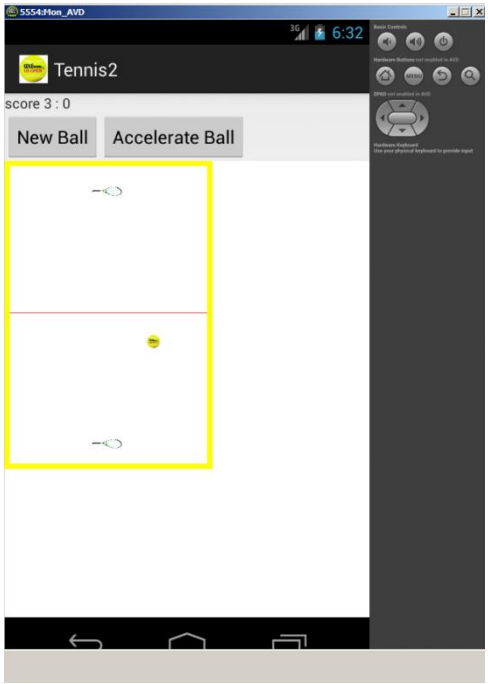
```
public class Balle {
   public RectF r;
   private int color;
   private int dx, dy;
   DrawView d;

   public Balle (int x1, int y1, DrawView d) {
      r = new RectF(x1, y1, x1+30, y1+30);
```

```
      this.color =
Color.argb((int)(255*Math.random()),(int)(255*Math.random()),

(int)(255*Math.random()),(int)(255*Math.random()));
      this.d = d;
      dx = 1;
      dy = 1;
   }

   public void dessineToi(Canvas canvas){
      d.paint.setColor(color);
      canvas.drawOval(r,d.paint);
   }

   public void changeDirection(boolean x)
   {
      if (x)
      {
         this.dy = -dy;
      }
      else
      {
         this.dx = -dx;
      }
      r.offset(5 * dx, 5 * dy);
   }

   public void bouge(Parois[] lesParois, ArrayList<Balle> lesBalles) {
      r.offset(dx, dy);
      for (Parois p : lesParois)
      {
         p.gereBalle(this);
      }

      for (Balle b : lesBalles) {
         if ((this != b) && RectF.intersects(r,  b.r)) {
            b.rebondit();
            rebondit();
         }
      }
   }

   public void rebondit() {
      dx = -dx;
      dy = -dy;
      r.offset(3 * dx, 3 * dy);
   }

}

public class Parois {
   private RectF r;
```

```
    private DrawView d;

    public Parois (int x1, int y1, int x2, int y2, DrawView d) {
        r = new RectF(x1, y1, x2, y2);
        this.d = d;
    }

    public void changeDimension(int nx1, int ny1, int nx2, int ny2){
        r.left = nx1;
        r.top=ny1;
        r.right=nx2;
        r.bottom = ny2;
    }


    public void dessineToi(Canvas canvas){
        d.paint.setColor(Color.GREEN);
        canvas.drawRect(r, d.paint);
    }

    public void gereBalle(Balle b) {
        if (RectF.intersects(r, b.r))
        {
            if (r.width()> r.height())
            {
                b.changeDirection(true);
            }
            else
            {
                b.changeDirection(false);
            }
        }
    }
}
```

## Application Tennis

Passons maintenant au « Tennis » représenté ci-dessous, également dans son émulateur Android. Le bouton de gauche lance la balle, celui de droite l'accélère. La raquette se déplace de gauche à droite à l'aide de la souris. Le score est affiché dans la zone texte du haut.




J'ai, pour raccourcir la présentation, écrit mon code dans seulement deux fichiers : `MainActivity.java` et `DrawView.java` ; la bonne programmation objet incite à coder chaque classe dans son propre fichier.

Passons d'abord le premier fichier en revue.

**MainActivity.java**

```
public class MainActivity extends Activity  {
   DrawView drawView;
   Button b1, b2;
   TextView tv;

   protected static final int GUIUPDATEIDENTIFIER = 0;

   Handler myGUIUpdateHandler = new Handler() {

      @Override
      public void handleMessage(Message msg){
         switch (msg.what){
         case MainActivity.GUIUPDATEIDENTIFIER:
            drawView.update();
            tv.setText("score " +
drawView.raq[0].getScore()+" : " +
                  drawView.raq[1].getScore());
```

```
            drawView.invalidate();
            super.handleMessage(msg);
            break;
         }

      }
   };

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      b1= (Button) findViewById(R.id.button);
      b2= (Button) findViewById(R.id.accelerate);
      b1.setOnClickListener( new OnClickListener() {
         public void onClick(View v) {
            Balle b = new Balle(drawView, drawView.raq,
drawView.bords,drawView.bmp);
            drawView.setBall(b);
            for (Raquette r : drawView.raq){
               r.setBalle(b);
            }
            for (Bord bo : drawView.bords){
               bo.setBalle(b);
            }
         }
      });
      b2.setOnClickListener( new OnClickListener() {
         public void onClick(View v) {
            drawView.b.accelerate();
         }
      });

      tv = (TextView) findViewById(R.id.score);
      drawView = (DrawView) findViewById(R.id.vMain);
      drawView.setBackgroundColor(Color.WHITE);
      new Thread (new RefreshRunner()).start();

   }


   class RefreshRunner implements Runnable {
      // @Override
      public void run() {
         while (!Thread.currentThread().isInterrupted()) {
            Message message = new Message();
            message.what =
MainActivity.GUIUPDATEIDENTIFIER;

MainActivity.this.myGUIUpdateHandler.sendMessage(message);

            try {
```

```
                Thread.sleep(10);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

La partie « multithreading » est en tout point semblable à celle décrite dans le code précédent. La grande nouveauté dans ce code-ci est l'utilisation des deux boutons et du champ texte. Dans la tradition de la séparation Model-View-Control, caractéristique de tout projet informatique, le développement sous Android encourage de séparer l'apparence de l'application, écrite de manière déclarative sous forme XML, et la partie contrôle inhérente à ces composants (boutons, champ texte…) écrite, elle, en Java dans les fichiers sources.

Ainsi, voici la description XML de l'application Tennis.

**Description XML de l'application Tennis**

```
<TextView android:text="Score: "
                android:id="@+id/score"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentLeft="true">
 </TextView>

<RelativeLayout
     android:layout_width="wrap_content"
     android:layout_height="wrap_content">

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="New Ball" />

 <Button
     android:id="@+id/accelerate"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_toRightOf="@id/button"
     android:text="Accelerate Ball"
    />

 </RelativeLayout>

<com.example.tennis2.DrawView
```

```
    android:id="@+id/vMain"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</LinearLayout>
```

On y trouve d'abord le champ « texte » pour afficher le score puis, ensemble, les deux boutons, et finalement le `DrawView` dans lequel apparaît le tennis à proprement parler. La description XML de cet affichage (notamment les `RelativeLayout`) nécessiterait un approfondissement de la programmation Android qui n'est pas de mise ici. Les deux boutons sont pris en charge par les deux méthodes `onClick`. Le rôle du premier consiste à lancer une nouvelle balle et celui du deuxième à accélérer la vitesse de la balle.

Voyons maintenant le deuxième fichier, `DrawView.java`.

**Fichier DrawView.java**

```
abstract class ObjetGraphique {
   protected DrawView d;
   public ObjetGraphique (DrawView d) {
      this.d = d;
   }
   abstract public void draw (Canvas g);
}

class Filet extends ObjetGraphique{
   public Filet(DrawView d) {
      super(d);
   }
   public void draw (Canvas g) {
      d.paint.setColor(Color.RED);
      g.drawLine(10, 300, 400, 300, d.paint);
   }
}

class Raquette extends ObjetGraphique {
   public RectF r;
   private Balle b;
   private Bitmap bmp;
   private boolean pressed;
   private int dx;
   private int score;


   public Raquette (DrawView d, int posx, int posy, Bitmap
b) {
      super(d);
      r = new RectF(posx, posy, posx+60, posy + 20);
      this.bmp=b;
```

```
      pressed = false;
      score = 0;
      dx = 5;
   }


   public int getScore() {
      return score;
   }

   public void incrementeScore() {
      score++;
   }

   public void setBalle(Balle b) {
      this.b = b;
   }

   public void touchBalle() {
      if (RectF.intersects(r,b.r)) {
         b.changeDirectionVerticale();
      }
   }

   public void draw (Canvas g) {
      g.drawBitmap(bmp,null,r,null);
   }
}

class Bord extends ObjetGraphique {
   private RectF r;
   private Balle b;
   private Raquette[] raq;

   public Bord(DrawView d, int posx, int posy, int width,
int height, Raquette[] raq) {
      super(d);
      r = new RectF(posx, posy, posx+width, posy+height);
      this.raq = raq;
   }

   public void setBalle(Balle b) {
      this.b = b;
   }

   public void touchBalle() {
      if (b != null) {
         if (RectF.intersects (r,b.r)){
            if (r.width() < 50) {
               b.changeDirectionHorizontale();
            }
            else {
```

```
                b.stopBalle();
                if (r.top < 300)
                   raq[0].incrementeScore();
                else raq[1].incrementeScore();
                b = null;
             }
          }
       }
    }

    public void draw (Canvas g) {
       d.paint.setColor(Color.YELLOW);
       g.drawRect(r, d.paint);
    }
}

class Balle extends ObjetGraphique  {
    public static int vitx;
    public static int vity;
    public RectF r;
    private Bitmap bmp;
    public Raquette[] lesRaquettes;
    public Bord[] lesBords;

    public Balle (DrawView d, Raquette[] lesRaquettes,
Bord[] lesBords, Bitmap b) {
       super(d);
       int x = (int)(300*Math.random());
       int y = 100 + (int)(200 * Math.random());
       r = new RectF (x,y,x+30,y+30);
       this.lesRaquettes = lesRaquettes;
       this.lesBords = lesBords;
       bmp = b;
       vitx = 1;
       if (Math.random() > 0.5) vity = 1;
       else vity = - 1;
    }

    public void draw (Canvas g) {
       g.drawBitmap(bmp,null,r,null);
    }


    public void stopBalle() {
       vitx = 0;
       vity = 0;
    }

    public void changeDirectionVerticale() {
       vity = -vity;
       r.top += vity;
    }
```

```
   public void changeDirectionHorizontale() {
      vitx = - vitx;
      r.bottom += vitx;
   }

   public void accelerate() {
      vitx = 2*vitx;
      vity = 2*vity;
   }

   public void move () {
      r.offset(vitx, vity);
      for (Raquette r : lesRaquettes) {
         r.touchBalle();
      }
      for (Bord b : lesBords) {
         b.touchBalle();
      }
   }
}


public class DrawView extends View implements
OnTouchListener {
   public Bitmap bmp;
   public Paint paint;
   public TextView tv;
   public int canvasW = 500;
   public int canvasH = 600;
   public Button btn;
   private ObjetGraphique[] lesObjets
   = new ObjetGraphique[8];
   public Balle b;
   public Raquette raq[] = new Raquette[2];
   public Bord bords[] = new Bord[4];
   private Filet f;
   public int raqImage;
   public int balImage;


   public DrawView(Context context, AttributeSet attrs) {
      super(context, attrs);
      paint = new Paint();
      this.getResources();
      raqImage =
this.getResources().getIdentifier("racket", "drawable",
this.getContext().getPackageName());
      balImage =
this.getResources().getIdentifier("ic_launcher",
"drawable", this.getContext().getPackageName());
```

```
      Bitmap b =
BitmapFactory.decodeResource(getResources(), raqImage);
      bmp = BitmapFactory.decodeResource(getResources(),
balImage);
      raq[0] = new Raquette(this, 20, 50, b);
      lesObjets[0] = raq[0];
      raq[1] = new Raquette(this, 20, 550, b);
      lesObjets[1] = raq[1];
      bords[0] = new Bord(this,0,0, 400, 10, raq);
      lesObjets[2] = bords[0];
      bords[1] = new Bord(this, 400,0, 10, 610, raq);
      lesObjets[3] = bords[1];
      bords[2] = new Bord(this, 0,600, 400, 10, raq);
      lesObjets[4] = bords[2];
      bords[3] = new Bord(this, 0,0, 10, 600, raq);
      lesObjets[5] = bords[3];
      lesObjets[6] = new Filet(this);
      this.setOnTouchListener(this);

   }

   public void onDraw(Canvas canvas) {
      for (ObjetGraphique o : lesObjets) {
         if (o != null) {
            o.draw(canvas);
         }
      }
   }

   public boolean onTouch(View v, MotionEvent e) {
      switch (e.getAction()) {
      case MotionEvent.ACTION_DOWN:
      case MotionEvent.ACTION_MOVE:
      case MotionEvent.ACTION_UP:
         int x = (int)e.getRawX() - 70;
         if (x < canvasW - 120) {
            raq[0].r.offsetTo(x, raq[0].r.top);
            raq[1].r.offsetTo(x, raq[1].r.top);
         }
      }
      return true;
   }

   public void setBall(Balle b) {
      this.b = b;
      lesObjets[7] = b;
   }

   public void update() {
      if (b != null) {
         b.move();
      }
```

```
    }
}
```

Le code ressemble beaucoup à celui du chapitre 20 à quelques écarts près dus à la différence de certaines API Android. Il en est ainsi de la présence du `OnTouchListener` pour le déplacement des deux raquettes à l'aide de la souris et de la récupération des images de la balle et des raquettes.

L'exécution de ces deux codes requiert l'installation dans leurs répertoires respectifs des layout XML associés (pour la disposition et le graphisme), des images et de l'écriture des fichiers manifestes (également sous une forme XML). Le tout peut s'archiver dans un exécutable `.apk` qui se lancera sur n'importe quelle plate-forme Android.

## Jeu Canon

**Crédits**

Ce troisième code Android est très largement inspiré d'une des applications présentes dans la deuxième édition de l'excellent ouvrage de la famille Deitel, *Android for Programmers*.

Paul Deitel, Harvey Deitel et Abbey Deitel, *Android for Programmers: An App-Driven Approach*, 2e édition, Prentice Hall, 2014

Bien que le jeu dont le code est repris ci-dessous soit une parfaite réplique de celui présenté dans l'ouvrage original, je me suis permis une réécriture dans un esprit beaucoup plus orienté objet. Je distingue donc les classes `BalleCannon`, `Cannon`, `Cible` et `Obstacle`, qui ne l'étaient pas à l'origine.

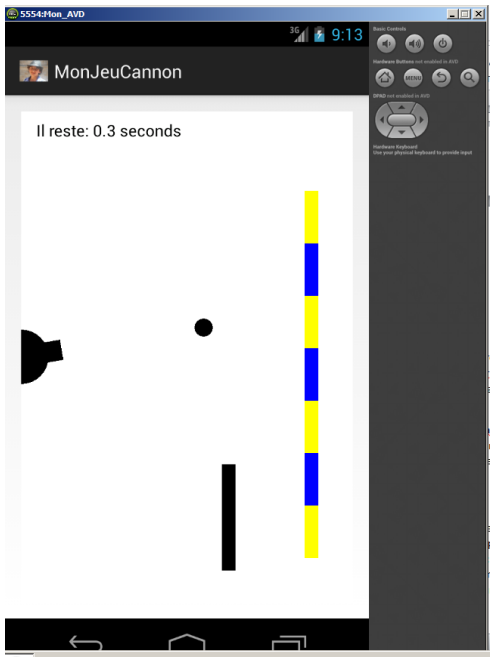
Le code d'origine est téléchargeable à l'adresse suivante (chapitre « Cannon Game ») :

http://www.deitel.com/Books/Android/AndroidforProgrammers2e/tabid/3653/Default.aspx

Il vous sera sans doute utile de vous rendre sur le site, notamment pour vous procurer les fichiers nécessaires à la partie sonore du jeu.

Nous remercions Harvey Deitel pour son aimable autorisation.

**MonJeuCanon**, comme la figure qui suit l'illustre, se déroule de la manière suivante : un canon tire des balles censées détruire les cibles qui se déplacent de haut en bas sur la droite. Un obstacle placé entre le canon et les cibles et qui se déplace aussi de haut en bas rend la chose plus compliquée (car les balles rebondissent sur l'obstacle). Le joueur a 10 secondes pour détruire toutes les cibles. Lorsqu'il touche l'obstacle, le temps qui lui reste diminue, et à chaque cible abattue, il gagne un peu de temps bonus. Des sons sont émis aux tirs du canon, aux chocs sur l'obstacle et aux chocs sur les cibles.




Le multithreading est géré de manière plus moderne et plus carrée que dans les deux codes qui précèdent. Il est fait appel à la classe `SurfaceView` pour le graphisme et l'interface `SurfaceHolder.Callback` pour la gestion du multithread.

Commençons par la classe la plus simple, à nouveau `MainActivity`.

**La classe MainActivity**

```
public class MainActivity extends Activity {

      private CannonView cannonView; // la view pour
afficher le jeu
      @Override
      protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_main);
         cannonView = (CannonView)
findViewById(R.id.vMain);
      }
}
```

Puis viennent les cinq classes essentielles au scénario repris dans ce jeu.

**Classes BalleCannon, Cannon, Cible, Line et Obstacle**

```
public class BalleCannon { // La balle du canon

   public Point cannonball;
   private int cannonballVitesseX;
   private int cannonballVitesseY;
   public boolean cannonballOnScreen;
   private int cannonballRadius;
   private int cannonballVitesse;
   private Paint cannonballPaint;
   private Obstacle obstacle;
   private Cible cible;
   private CannonView view;

   public BalleCannon(CannonView view, Obstacle obstacle,
Cible cible) {
         cannonball = new Point();
         cannonballPaint = new Paint();
         this.view = view;
         this.obstacle = obstacle;
         this.cible = cible;
   }

   public void launch(double angle) {
         cannonball.x = cannonballRadius;
         cannonball.y = view.screenHeight / 2;
         cannonballVitesseX = (int) (cannonballVitesse *
Math.sin(angle));
         cannonballVitesseY = (int) (-cannonballVitesse *
Math.cos(angle));
         cannonballOnScreen = true;
   }

   public void setCannonballRadius (int radius) {
         cannonballRadius = radius;
   }

   public void draw(Canvas canvas) {
         canvas.drawCircle(cannonball.x, cannonball.y,
cannonballRadius,
                  cannonballPaint);
   }

   public void setCannonballVitesse(int vitesse) {
         cannonballVitesse = vitesse;
   }

   public void resetCannonBall() {
         cannonballOnScreen = false;
   }
```

```
   public void update(double interval) {
         if (cannonballOnScreen)
            {

               cannonball.x += interval *
cannonballVitesseX;
               cannonball.y += interval *
cannonballVitesseY;

               if (cannonball.x + cannonballRadius >
obstacle.getObstacleDistance() &&
                  cannonball.x - cannonballRadius <
obstacle.getObstacleDistance() &&
                  cannonball.y + cannonballRadius >
obstacle.getLine().start.y &&
                  cannonball.y - cannonballRadius <
obstacle.getLine().end.y)
               {
                  cannonballVitesseX *= -1;
                  view.reduceTimeLeft();
                  view.playObstacleSound();
               }
           else if (cannonball.x + cannonballRadius >
view.screenWidth ||
                  cannonball.x - cannonballRadius < 0)
               {
                  cannonballOnScreen = false;
               }
               else if (cannonball.y + cannonballRadius >
view.screenHeight ||
                  cannonball.y - cannonballRadius < 0)
               {
                  cannonballOnScreen = false;
               }
               else if (cannonball.x + cannonballRadius >
cible.getCibleDistance() &&
                  cannonball.x - cannonballRadius <
cible.getCibleDistance() &&
                  cannonball.y + cannonballRadius >
cible.getLine().start.y &&
                  cannonball.y - cannonballRadius <
cible.getLine().end.y)
               {
              cible.detectChoc(this);
               }
            }
   }
}

public class Cannon { // le canon qui tire les balles
       private int cannonBaseRadius;
       private int cannonLongueur;
```

```
       private Point finCannon;
       private Paint cannonPaint;
       private int width;
       private CannonView view;

       public Cannon(CannonView view) {
          cannonPaint = new Paint();
          this.view = view;
       }

       public void draw (Canvas canvas) {
          canvas.drawLine(0, view.screenHeight / 2,
finCannon.x,finCannon.y,
                  cannonPaint);
          canvas.drawCircle(0, (int) view.screenHeight / 2,
                  (int) cannonBaseRadius, cannonPaint);
       }

       public void align(double angle) {
          finCannon.x = (int) (cannonLongueur *
Math.sin(angle));
           finCannon.y = (int) (-cannonLongueur *
Math.cos(angle) + view.screenHeight / 2);
       }

       public void setCannonBaseRadius(int radius) {
          cannonBaseRadius = radius;
       }

       public void setWidth(int width) {
          this.width = width;
          cannonPaint.setStrokeWidth(width * 1.5f);
       }

       public void setCannonLongueur(int length) {
          cannonLongueur = length;
       }

       public void setFinCannon(int x) {
          finCannon = new Point(cannonLongueur, x);
       }
}

public class Cible { // La cible faite de 7 rectangles de
couleur alternée
      public static final int CIBLE_PIECES = 7;
      private Line cible;
      private int cibleDistance;
      private int cibleDebut;
      private double cibleLongueur;
      private int cibleFin;
      private int cibleVitesseInitiale;
```

```
      private float cibleVitesse;
      private int longueurPiece;
       private int largeur;
      private boolean[] cibleTouchee;
      private int nbreCiblesTouchees;
      private Paint ciblePaint;
      private int width;
      private CannonView view;

      public Cible(CannonView view) {
         cible = new Line();
         cibleTouchee = new boolean[CIBLE_PIECES];
         ciblePaint = new Paint();
         this.view = view;
      }

      public void detectChoc(BalleCannon balle) {
           int section =  (int) ((balle.cannonball.y-
cible.start.y) / longueurPiece);

           if ((section >= 0 && section < CIBLE_PIECES) &&
!cibleTouchee[section])
            {
                cibleTouchee[section] = true;
                balle.resetCannonBall();
                view.increaseTimeLeft();
                view.playCibleSound();

                 if (++nbreCiblesTouchees == CIBLE_PIECES)
                 {
                      view.gameOver();
                 }
             }
      }

      public void update(double interval) {
         double up = interval * cibleVitesse;
          cible.start.y += up;
          cible.end.y += up;
          if (cible.start.y < 0 || cible.end.y >
view.screenHeight)
               cibleVitesse *= -1;
      }

      public void draw(Canvas canvas){
         Point currentPoint = new Point();
            currentPoint.x = cible.start.x;
            currentPoint.y = cible.start.y;
            for (int i = 0; i < CIBLE_PIECES; i++)
            {
               if (!cibleTouchee[i])
               {
```

```
                if (i % 2 != 0)
                   ciblePaint.setColor(Color.BLUE);
                else
                   ciblePaint.setColor(Color.YELLOW);

                canvas.drawLine(currentPoint.x,
currentPoint.y, cible.end.x,
                   (int) (currentPoint.y +
longueurPiece), ciblePaint);
             }

             currentPoint.y += longueurPiece;
          }
     }

     public int getCibleDistance() {
        return cibleDistance;
     }

     public Line getLine() {
        return cible;
     }

     public void setCibleDistance(int dist) {
        cibleDistance = dist;
     }

     public void resetCible() {
         for (int i = 0; i < CIBLE_PIECES; i++)
               cibleTouchee[i] = false;
         nbreCiblesTouchees = 0;
         cibleVitesse = cibleVitesseInitiale;
         cible.start.set(cibleDistance, cibleDebut);
            cible.end.set(cibleDistance, cibleFin);
     }

     public void setCibleDebut(int deb) {
        cibleDebut = deb;
        cible.start = new Point(cibleDistance, deb);
     }

     public void setCibleVitesseInitiale(int vit) {
        cibleVitesseInitiale = vit;
     }

     public void setCibleFin(int fin){
        cibleFin = fin;
        cible.end = new Point(cibleDistance, fin);
        longueurPiece = (cibleFin -
cibleDebut)/CIBLE_PIECES;
     }
```

```
      public void setWidth(int width) {
            this.width = width;
            ciblePaint.setStrokeWidth(width);
      }
}

public class Line
{
   public Point start = new Point();
   public Point end = new Point();
}

public class Obstacle { // L'obstacle qui fait rebondir la
balle
         private Line obstacle;
         private int obstacleDistance;
         private int obstacleDebut;
         private int obstacleFin;
         private int initialObstacleVitesse;
         private float obstacleVitesse;
         private Paint obstaclePaint;
         private int width;
         private CannonView view;

         public Obstacle(CannonView view) {
            obstaclePaint = new Paint();
            obstacle = new Line();
            this.view = view;
         }

         public int getObstacleDistance() {
            return obstacleDistance;
         }

         public Line getLine() {
            return obstacle;
         }

         public void update(double interval) {
            double up = interval * obstacleVitesse;
            obstacle.start.y += up;
            obstacle.end.y += up;
            if (obstacle.start.y < 0 || obstacle.end.y >
view.screenHeight)
                  obstacleVitesse *= -1;
         }

         public void draw(Canvas canvas) {
canvas.drawLine(obstacle.start.x, obstacle.start.y,
obstacle.end.x,obstacle.end.y, obstaclePaint);
         }
```

```
        public void setWidth(int width) {
           this.width = width;
           obstaclePaint.setStrokeWidth(width);
        }

        public void resetObstacle() {
          obstacleVitesse = initialObstacleVitesse;
           obstacle.start.set(obstacleDistance,
obstacleDebut);
              obstacle.end.set(obstacleDistance,
obstacleFin);
        }

        public void setObstacleDistance (int dist) {
           obstacleDistance = dist;
        }

        public void setObstacleDebut(int deb) {
           obstacleDebut = deb;
           obstacle.start = new Point(obstacleDistance,
deb);
        }

        public void setObstacleFin( int fin) {
           obstacleFin = fin;
           obstacle.end = new Point(obstacleDistance,
fin);
        }

        public void seInitialObstacleVitesse(int vit) {
           initialObstacleVitesse = vit;
        }

}
```

La classe la plus intéressante est sans doute celle qui suit. C'est la gestion du thread qui demande le plus d'attention. Dans la méthode `run` du thread, l'animation se redessine continûment sur un objet `Canvas` qui est bloqué le temps de chaque rafraîchissement. Cette manière de programmer la partie multithread est légèrement plus compliquée que celle décrite pour les deux applications précédentes, mais elle fait appel à des librairies plus récentes de l'univers Android.

Les trois méthodes `SurfaceCreated`, `Changed` et `Destroyed` proviennent de l'interface `SurfaceHolder.Callback` et s'occupent de la gestion synchronisée du canvas.

La classe `CannonView` hérite de la classe `SurfaceView` qui est prévue pour, de fait, s'occuper des animations graphiques possédant leur propre thread d'exécution.

**La classe CannonView**

```
public class CannonView extends SurfaceView
   implements SurfaceHolder.Callback
{
   private static final String TAG = "CannonView"; // for
logging errors

   private CannonThread cannonThread; // contrôle la boucle
principale du jeu
   private BalleCannon balle;
   private Cannon cannon;
   private Cible cible;
   private Obstacle obstacle;
   private Activity activity; // afin de montrer le
dialogue Game Over
   private boolean dialogIsDisplayed = false;

   // les constantes essentielles du jeu

   public static final int MISS_PENALTY = 2; // les
secondes déduites en cas de raté
   public static final int HIT_REWARD = 3; // les seconds
ajoutées en cas de succès.

   // variables pour la boucle principale et pour suivre
l'évolution du jeu.
   private boolean gameOver; // game over?
   private double timeLeft; // temps restant en secondes
   private int shotsFired; // nombres de tirs effectués par
le joueur
   private double totalElapsedTime; // temps écoulé
   public int screenWidth;
   public int screenHeight;

   // variables et constantes pour la gestion du son
   private static final int TARGET_SOUND_ID = 0;
   private static final int CANNON_SOUND_ID = 1;
   private static final int BLOCKER_SOUND_ID = 2;
   private SoundPool soundPool; // effet sonore
   private SparseIntArray soundMap; // colle l'ID au pool
sonore

   // les variables utilisées pour dessiner chaque élément
du jeu
   private Paint textPaint; // le Paint utilisé pour le
texte to draw text
   private Paint backgroundPaint; // le Paint utilisé pour
redessiner le fond

   // le Constructeur
   public CannonView(Context context, AttributeSet attrs)
```

```
   {
      super(context, attrs);
      activity = (Activity) context; // la référence à la
l'Activité principale

      // l'écoute du SurfaceHolder.Callback
      getHolder().addCallback(this);

      // initialisation de tous les objets du jeu
      obstacle = new Obstacle(this);
      cible = new Cible(this);
      cannon = new Cannon(this);
      balle = new BalleCannon(this,obstacle, cible);

      // initialisation du SoundPool pour jouer les trois
effets sonores
      soundPool = new SoundPool(1,
AudioManager.STREAM_MUSIC, 0);

      // crée la Map de sounds et précharge les sons
      soundMap = new SparseIntArray(3); // crée un nouvel
HashMap
      soundMap.put(TARGET_SOUND_ID,
         soundPool.load(context, R.raw.target_hit, 1));
      soundMap.put(CANNON_SOUND_ID,
         soundPool.load(context, R.raw.cannon_fire, 1));
      soundMap.put(BLOCKER_SOUND_ID,
         soundPool.load(context, R.raw.blocker_hit, 1));
      textPaint = new Paint();
      backgroundPaint = new Paint();
   }

   // Appelé par surfaceChanged quand la dimension de la
SurfaceView change,
   // Et aussi lors du premier appel
   @Override
   protected void onSizeChanged(int w, int h, int oldw, int
oldh)
   {
      super.onSizeChanged(w, h, oldw, oldh);

      screenWidth = w; // largeur de l'écran
      screenHeight = h; // hauteur
      cannon.setCannonBaseRadius(h/18); rayon du canon
      cannon.setCannonLongueur(w / 8); // longueur
      balle.setCannonballRadius(w / 36); // rayon de la
balle
      balle.setCannonballVitesse(w * 3 / 2); //
multiplicateur de vitesse
      cible.setWidth(w/24);
      obstacle.setWidth(w / 24); // largeur des obstacles
      cannon.setWidth(w/24);
```

```
            obstacle.setObstacleDistance(w * 5 / 8);
            // distance des obstacles
            obstacle.setObstacleDebut(h / 8);
            obstacle.setObstacleFin(h * 3 / 8);
            obstacle.seInitialObstacleVitesse(h / 2);
            // vitesse initiale des obstacles


      cible.setCibleDistance(w * 7 / 8);
      // distance de la cible
      cible.setCibleDebut(h / 8);
      cible.setCibleFin(h * 7 / 8);
      cible.setCibleVitesseInitiale(-h / 4);
      // vitesse initiale de la cible


      cannon.setFinCannon(h / 2);

      textPaint.setTextSize(w / 20); // taille du texte
      textPaint.setAntiAlias(true);
      backgroundPaint.setColor(Color.WHITE); // couleur de
fond

      newGame(); // commence un nouveu jeu
   }

   public void reduceTimeLeft() {
         timeLeft -= MISS_PENALTY;
   }

   public void increaseTimeLeft() {
         timeLeft += HIT_REWARD;
   }

   public void playObstacleSound() {
         soundPool.play(soundMap.get(BLOCKER_SOUND_ID), 1,
1, 1, 0, 1f);
   }

   public void playCibleSound() {
         soundPool.play(soundMap.get(TARGET_SOUND_ID), 1,
               1, 1, 0, 1f);
   }
   // reset tous les éléments de l'écran et démarre un
nouveau jeu
   public void newGame()
   {

      cible.resetCible();
      obstacle.resetObstacle();
```

```
      timeLeft = 10; // commence le décompte à 10 secondes
      balle.resetCannonBall(); // la balle n'est pas sur
l'écran
      shotsFired = 0; // initialise le nombre de tirs
      totalElapsedTime = 0.0; // initialise le temps écoulé

      if (gameOver) // commence un nouveau jeu
      {
         gameOver = false;
         cannonThread = new CannonThread(getHolder());
         // création du thread
         cannonThread.start(); // démarage du thread
      }
   }

   public void gameOver() {
       cannonThread.setRunning(false); // fin du thread
        showGameOverDialog(R.string.win); // le dialogue de
la victoire
        gameOver = true;
   }

   // mise à jour des éléments
   private void updatePositions(double elapsedTimeMS)
   {
      double interval = elapsedTimeMS / 1000.0;
      // conversion en secondes
      balle.update(interval);


      obstacle.update(interval);


      cible.update(interval);

      timeLeft -= interval;
      // si le timer atteint 0
      if (timeLeft <= 0.0)
      {
         timeLeft = 0.0;
         gameOver = true; // game over
         cannonThread.setRunning(false); // fin du thread
         showGameOverDialog(R.string.lose); // montre le
dialogue de la défaite
      }
   }

   // tire une balle
   public void fireCannonball(MotionEvent event)
   {
      if (balle.cannonballOnScreen)         return;
```

```
      double angle = alignCannon(event); // obtient l'angle
du canon

      // la balle doit être dans le canon.
      balle.launch(angle);
      ++shotsFired; // plus de tirs

      // le son du tir
      soundPool.play(soundMap.get(CANNON_SOUND_ID), 1, 1,
1, 0, 1f);
   }
   // réponse de la touche du joueur sur le canon
   public double alignCannon(MotionEvent event)
   {

      Point touchPoint = new Point((int) event.getX(),
(int) event.getY());


      double centerMinusY = (screenHeight / 2 -
touchPoint.y);

      double angle = 0;
      // calcul de l'angle fait avec l'horizontal
      if (centerMinusY != 0) //         angle =
Math.atan((double) touchPoint.x / centerMinusY);


      if (touchPoint.y > screenHeight / 2)
         angle += Math.PI;

            cannon.align(angle);
      return angle;   }
   // dessine le jeu dans le Canvas
   public void drawGameElements(Canvas canvas)
   {
      // remet le background à 0
      canvas.drawRect(0, 0, canvas.getWidth(),
canvas.getHeight(),
         backgroundPaint);
      // montre le temps qui reste
      canvas.drawText(getResources().getString(
         R.string.time_remaining_format, timeLeft), 30, 50,
textPaint);
      // si la balle est sur l'écran, il faut la dessiner
      if (balle.cannonballOnScreen)
       balle.draw(canvas);
            cannon.draw(canvas);

      obstacle.draw(canvas);
            cible.draw(canvas);
```

```
   }

   // montre un dialogue alerte quand le jeu se termine
private void showGameOverDialog(final int messageId)
   {
            final DialogFragment gameResult =
         new DialogFragment()
         {
            // crée un AlertDialog
            @Override
            public Dialog onCreateDialog(Bundle bundle)
            {
                              AlertDialog.Builder builder =
                  new AlertDialog.Builder(getActivity());

builder.setTitle(getResources().getString(messageId));

               // montre le nombre de tirs et le temps
écoulé
               builder.setMessage(getResources().getString(
                  R.string.results_format, shotsFired,
totalElapsedTime));

builder.setPositiveButton(R.string.reset_game,
                  new DialogInterface.OnClickListener()
                  {
                     // appelé quand le bouton reset est
pressé
                     @Override
                     public void onClick(DialogInterface
dialog, int which)
                     {
                        dialogIsDisplayed = false;
                        newGame(); // un nouveau jeu.
                     }
                  }
               );

               return builder.create();            }
         };
      // dans le Gui principal, on utilise le
FragmentManager pour montrer le DialogFragment
      activity.runOnUiThread(
         new Runnable() {
            public void run()
            {
               dialogIsDisplayed = true;
               gameResult.setCancelable(false);
gameResult.show(activity.getFragmentManager(), "results");
            }
         }
      );
```

```
   }
   // arrête le jeu ; appelé par la méthode
CannonGameFragment's onPause
   public void stopGame()
   {
      if (cannonThread != null)
         cannonThread.setRunning(false); // dit au thread
de s'arrêter
   }

   // libère les ressources ; appelé par la méthode
CannonGameFragment's onDestroy
   public void releaseResources()
   {
      soundPool.release();      soundPool = null;
   }

   // appelé par des changements de taille de la surface
   @Override
   public void surfaceChanged(SurfaceHolder holder, int
format,
      int width, int height)
   {
   }

   // appelé quand la surface est créée pour la première
fois
   @Override
   public void surfaceCreated(SurfaceHolder holder)
   {
      if (!dialogIsDisplayed)
      {
         cannonThread = new CannonThread(holder);
         // un nouveau thread
         cannonThread.setRunning(true);
         // démarrage du Thread
         cannonThread.start();      }
   }

   // appelé quand la surface est détruite
   @Override
   public void surfaceDestroyed(SurfaceHolder holder)
   {
      // terminaison correct du thread
      boolean retry = true;
      cannonThread.setRunning(false);
      while (retry)
      {
         try
         {
            cannonThread.join();
            // attendre que le cannonThread se termine
```

```
            retry = false;
         }
         catch (InterruptedException e)
         {
            Log.e(TAG, "Thread interrupted", e);
         }
      }
   }
   // appelé quand le joueur touche l'écran
   @Override
   public boolean onTouchEvent(MotionEvent e)
   {
      // quelle action ?
      int action = e.getAction();

            if (action == MotionEvent.ACTION_DOWN ||
         action == MotionEvent.ACTION_MOVE)
      {
         fireCannonball(e); // tire la balle
      }

      return true;
   }

   // le thread contrôlant la boucle principale
   private class CannonThread extends Thread
   {
      private SurfaceHolder surfaceHolder;
      // pour le canvas
      private boolean threadIsRunning = true;
      // initialise le surface holder
      public CannonThread(SurfaceHolder holder)
      {
         surfaceHolder = holder;
         setName("CannonThread");
      }

            public void setRunning(boolean running)
      {
         threadIsRunning = running;
      }

      // contrôle la boucle principale
      @Override
      public void run()
      {
         Canvas canvas = null; // pour dessiner
         long previousFrameTime =
System.currentTimeMillis();

         while (threadIsRunning)
         {
```

```
            try
            {
               // obtient le canvas pour le dessin
               canvas = surfaceHolder.lockCanvas(null);

               // bloque le surfaceHolder pour dessiner
               synchronized(surfaceHolder)
               {
                  long currentTime =
System.currentTimeMillis();
                  double elapsedTimeMS = currentTime -
previousFrameTime;
                  totalElapsedTime += elapsedTimeMS /
1000.0;
                  updatePositions(elapsedTimeMS);
                  // mise à jour du jeu
                  drawGameElements(canvas);
                  // dessine via la canvas
                  previousFrameTime = currentTime;
                  // update le temps
               }
            }
            finally
            {
               // montre le canvas et autorise d'autres
               // threads a utiliser le canvas
               if (canvas != null)

surfaceHolder.unlockCanvasAndPost(canvas);
            }
         }
      }
   }
}
```