

# **Android Balls!**

Jordi Fita

**COLLABORATORS**

	<i>TITLE :</i> Android Balls!		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jordi Fita	September 10, 2022	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME
34b7522b4f97	2011-03-28	atangle is now using a new style for directives which don't collide with XML tags. I had to update all games and programs as well in order to use the new directive syntax.	jfita
6cc909c0b61d	2011-03-07	Added the comments section.	jfita
76ad1e99b212	2011-02-10	Fixed some typos.	jfita
065ebf817c7b	2011-01-28	The constants are static now.	jfita
583c9b6fc2cc	2011-01-25	Added the download for balls-android.	jfita
b27524efe59d	2011-01-25	Cleaned up the text and made it somewhat more understandable.	jfita
06e8b46be171	2011-01-24	Fixed a bug in setting the direction while updating the balls.	jfita
e51254c1426c	2011-01-24	Finished the first draft. The game works now.	jfita
61d9f37d5bd1	2011-01-24	Added the balls.	jfita
e2f01597bac3	2011-01-23	Added the levels.	jfita

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME
f06c5329b85f	2011-01-23	Added the BallsActivity class and BallsView initialization. Fixed problem with layout's schema.	jfita
b3601a99ce26	2011-01-23	Added the main game's layout.	jfita
ace67aa254c8	2011-01-22	Added the strings file.	jfita
d550065cace8	2011-01-22	Added AndroidManifest.xml	jfita
786b90b622c0	2011-01-22	Added the GPL license to Android's Balls.	jfita
2206ad2a6d25	2011-01-22	Added the empty code for Android's Balls. I also added this project to the top's Makefile.	jfita
30c899f9f146	2011-01-22	Added the android target in ball's readme.	jfita
ba2dbb0d0445	2011-01-21	Added the introduction to android balls.	jfita

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Download . . . . .	1
<b>2</b>	<b>Code</b>	<b>1</b>
2.1	The Level's Layout . . . . .	1
2.2	Manifest . . . . .	2
2.3	Starting the Game . . . . .	3
2.4	The View . . . . .	4
2.4.1	Initialization . . . . .	4
2.4.2	The Levels . . . . .	5
2.4.3	Balls . . . . .	6
2.4.4	Explosions . . . . .	8
2.4.5	Updating the Game State . . . . .	11
2.4.6	Pushing the Pixels on the Screen . . . . .	13
2.4.7	Tap, Tap, Tapping . . . . .	13
2.4.8	BallsView.java . . . . .	14
2.5	Strings File . . . . .	14
<b>A</b>	<b>License</b>	<b>15</b>

# 1 Introduction

*Android Balls!* is an **Android** port of **Balls!**, a very simple puzzle game in which the player must remove an specific number of bouncing ball from the screen by starting a chain reaction of explosions. In this port, instead of using the mouse, the player must tap on the screen to start the first explosion and trigger the chain reaction.

Being an Android application, *Android Balls!* is written in Java. Here I target Android 2.2 (android-8) because that's the version my Android smartphone uses.

## 1.1 Download

There is no precompiled version of *Android Balls!* due to security concerns issues and because I don't believe this game is worthy to be listed in the **Android Market**. But, the source code, extracted with **atangle** and a Makefile to build the whole thing from the **AsciiDoc** document is available at the following URL:

<http://www.geishastudios.com/download/balls-android.zip>

Also, for those interested in this game's original AsciiDoc document, the latest version is always available at:

<http://dev.geishastudios.com/literate/src/tip/balls-android/>

# 2 Code

## 2.1 The Level's Layout

In Android, the easiest way to define the application's user interface is by using special XML files called *layout files*. Those layout files contains the user controls to use as well as special object whose purpose is to arrange the controls around the screen.

Being a simple game, *Balls!* only needs a single screen or layout that fills the whole screen. Furthermore, this layout has a single main control, called a view, which is the responsible to draw the game's elements and to receive user input. Thus, the best layout object for this game is the *FrameLayout* because this object contains only a single control within. If there there are more controls inside this layout, then *FrameLayout* simply stacks one over the previous overlapping the contents. This is useful, in this case, to make the view fill the screen and later add the texts that show the game's status.

```
<<balls.xml>>=
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="#2a547e"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <<balls view control>>
    <<status texts>>
</FrameLayout>
```

Notice how I even specified here the background color to use for the level. Although it is possible to specify the background color from code, I usually prefer to stick all user interface specific bits together.

Inside the *FrameLayout* I add the full screen view that I am going to use for the game. In this case, I am using a *View* derived class meaning that Android will ask this control to draw itself and will send the appropriate touch events when the user taps on the screen. Since this class does not belong to Android's namespace, I must specify the full name of the object.

```
<<balls view control>>=
<com.geishastudios.balls.BallsView
    android:id="@+id/level"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

After the View, I also add a pair of *TextView* controls. One is shown before the actual gameplay begins and asks the player to tap the screen to start the level. The other *TextView* is used during the gameplay to show the number of balls exploded and the target number of balls to explode.

As said before, adding additional controls on a *FrameLayout* would stack those controls on the top-left corner but I want to place these two *TextView* at different positions. To achieve that I group the *TextViews* in a *RelativeLayout* object which allows me to put the controls relatively to its parent. In this case, the game status *TextView* is aligned on the bottom-right corner while the other is centered on the screen.

```
<<status texts>>=
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/next_level"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:gravity="center_horizontal"
        android:text="@string/next_level"
        android:textColor="#ffffff"
        android:textSize="24sp" />
    <TextView
        android:id="@+id/status"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentBottom="true"
        android:textColor="#ffffff"
        android:textSize="20sp"
        android:visibility="invisible"/>
</RelativeLayout>
```

The text specified for *next\_level* is a reference of the string written in the strings file (see Section 2.5.)

```
<<press tab to start string>>=
<string name="next_level">Tap to Start Next Level</string>
```

## 2.2 Manifest

Every Android application must have a manifest file, named `AndroidManifest.xml`, in its root directory which has the essential information required by the system to run the application. This manifest file, as many of Android's resource files, is an XML file.

```
<<AndroidManifest.xml>>=
<?xml version="1.0" encoding="utf-8"?>
```

The root element of the manifest must be the `manifest` element. In this element, besides the required namespace, I need to specify the application's package and version.

```
<<AndroidManifest.xml>>=
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.geishastudios.balls"
    android:versionCode="1"
    android:versionName="1.0">
    <<application specification>>
</manifest>
```

As a child of manifest, there must be the application's name and icon. This is given as attributes of the application element.

```
<<application specification>>=  
<application  
    android:label="@string/app_name"  
    android:icon="@drawable/icon">  
    <<activity specification>>  
</application>
```

The application name is a reference to the string in the strings file (see Section 2.5.)

```
<<application name string>>=  
<string name="app_name">Balls</string>
```

Inside the `application` element I must define at least one activity for the application. In this case, being a simple game, there is only one activity: *BallsActivity*. The name of the `activity` element must be the same as the Java class that implements the activity.

```
<<activity specification>>=  
<activity  
    android:name="BallsActivity"
```

The label is just the text to show to the player while the activity is running.

```
<<activity specification>>=  
    android:label="@string/app_name"
```

For this activity, though, I am using a theme that has no title bar, thus the label won't be actually visible.

```
<<activity specification>>=  
    android:theme="@android:style/Theme.NoTitleBar"
```

Finally, I want to force a portrait screen orientation even when the user rotates the device or reveals the hardware keyboard, if any. For this to happen, I set the initial screen orientation to portrait and that the activity handles all changes referring to orientation or keyboard. In reality, I do nothing for these events, hence the orientation remains unchanged.

```
<<activity specification>>=  
    android:screenOrientation="portrait"  
    android:configChanges="keyboardHidden|orientation">
```

To close the activity, I must declare which messages, called intents, it can respond to. In this game there is only one activity which needs to be the initial activity at startup. This is accomplished by receiving the *MAIN* action intent.

```
<<activity specification>>=  
    <intent-filter>  
        <action android:name="android.intent.action.MAIN" />
```

And to be able to start this game from the application launcher it must also accept the *LAUNCHER* category.

```
<<activity specification>>=  
    <category android:name="android.intent.category.LAUNCHER" />  
    </intent-filter>  
</activity>
```

## 2.3 Starting the Game

To be able to run the game I need to write the game's sole activity that was specified in the manifest file: *BallsActivity*. In this case, the only thing I need to do is specify which layout the activity should use, the one declared in Section 2.1 above, and tell the view which text controls to use.

All this is done in the *onCreate* method overridden from the *Activity* class. This method is called by Android when an activity is created. In this case, being the startup activity, this *onCreate* method becomes, roughly speaking, the application's main function.

```
<<BallsActivity.java>>=
//
// Balls! -- A simple game with balls.
// Copyright 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
package com.geishastudios.balls;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class BallsActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.balls);

        BallsView view = (BallsView)findViewById(R.id.level);
        view.setNextLevelTextView((TextView)findViewById(R.id.next_level));
        view.setStatusTextView((TextView)findViewById(R.id.status));
    }
}
```

Notice how I am using the *R* class to get the references to the layout to use as well as the references for both *TextView* widgets. This *R* class is generated by *ant* when building the project from the definition of the resource files.

The *onCreate* parameter, an object of type *Bundle*, is usually used to restore the game to a previous state when the activity had to be suspended. For instance, when there is a phone call. For this game, I would store the level currently playing when the game is suspended and then restore the level from the *Bundle* object passed, but I didn't think it was worth for this game.

## 2.4 The View

The *BallsView* class could be considered a misnomer because this view, actually, is the main game's class that updates the screen as well as accepts input from the user (tapping) and acts on this input. Unfortunately, Android calls these classes *View* and thus, to use the existing nomenclature, I decided to call this class *BallsView*.

### 2.4.1 Initialization

This class is also referenced by the layout file above. This means that I don't need to manually instantiate this class as Android will do that for me once the layout where it is specified gets used. In this case, once the *BallsActivity* is created. But for this to happen, I must add the constructors expected by Android. For this class, the constructors only call the parent's constructors with the same signature.

```
<<BallsView constructors>>=
public BallsView(Context context, AttributeSet attrs) {
    super(context, attrs);
}

public BallsView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
}
```

I need to add the imports for the *Context* and *AttributeSet* classes.



```
<<BallsView imports>>=
import android.content.Context;
import android.util.AttributeSet;
```

Remember though, that besides the construction handled by Android, *BallsActivity* passes the references of the two *TextView* widgets that the view updates when starting the next level and while playing the game. These two can't be passed in the constructor because when *BallsActivity* sets its layout, the object is already constructed. Thus, I need to use additional methods for that.

```
<<setting the text views>>=
public void setNextLevelTextView(TextView textView) {
    nextLevelTextView = textView;
}

public void setStatusTextView(TextView textView) {
    statusTextView = textView;
}
```

These two variables are member attributes of *BallsView*.

```
<<BallsView attributes>>=
private TextView nextLevelTextView;
private TextView statusTextView;
```

And I need to import the correct package to use *TextView*.

```
<<BallsView imports>>=
import android.widget.TextView;
```

### 2.4.2 The Levels

*Balls!* levels only require two pieces of information: the minimum number of balls to explode — which I call target — and the number of initial balls on the screen. I store this information in a new *Level* class inside *BallsView* that has the two attributes and a constructor to set them.

```
<<Level class>>=
class Level {
    public final int balls;
    public final int target;

    Level(int balls, int target) {
        this.balls = balls;
        this.target = target;
    }
}
```

With this class declared, I can simply create an array as an attribute of *BallsView* where each value is a new instance of the class *Level* with the correct number of balls and target.

```
<<BallsView attributes>>=
private Level[] levels = {
    new Level(4, 1),
    new Level(4, 2),
    new Level(8, 4),
    new Level(15, 7),
    new Level(25, 12),
    new Level(30, 17),
    new Level(30, 20),
    new Level(30, 25),
}
```

```

        new Level(30, 28),
        new Level(30, 30)
    };

```

Of course, I need a way to know in which level I am currently on.

```

<<BallsView attributes>>=
private int currentLevel = 0;

```

### 2.4.3 Balls

The balls are the simplest of the objects. They get created when the level starts and move themselves following a direction and bouncing off when they reach the level's borders.

With this in mind, I create a simple class that asks for an initial position, the ball's radius, and color. The same constructors will set the ball to move towards the top left corner of the screen by initializing the direction to *-1* for both X and Y.

```

<<Ball class>>=
class Ball
{
    private Vector2D position;
    private Vector2D direction;
    private final float radius;
    private final Paint paint;

    public Ball(float x, float y, float radius, int color) {
        this.position = new Vector2D(x, y);
        this.direction = new Vector2D(-1, -1);
        this.radius = radius;
        this.paint = new Paint();
        this.paint.setColor(color);
    }

    public Vector2D getPosition() {
        return position;
    }

    public float getRadius() {
        return radius;
    }

    public int getColor() {
        return paint.getColor();
    }
}

```

With this constructor is possible to create the required balls for a level in random positions and with random colors with a simple for loop, storing the balls in an *ArrayList* to be used later when drawing or updating the game's state.

```

<<BallsView imports>>=
import java.util.ArrayList;
import java.util.Random;
import android.graphics.Color;

```

```

<<BallsView constants>>=
private static final float BALL_RADIUS = 4;

```

```

<<BallsView attributes>>=
private static final Random random = new Random();
private ArrayList<Ball> balls = new ArrayList<Ball>();
private int[] colors = {

```

```

    Color.RED,
    Color.GREEN,
    Color.BLUE,
    Color.CYAN,
    Color.YELLOW,
    Color.MAGENTA
};

```

```

<<create balls>>=
for(int ball = 0 ; ball < levels[currentLevel].balls ; ++ball ) {
    balls.add(new Ball(
        random.nextInt(getWidth()),
        random.nextInt(getHeight()),
        BALL_RADIUS,
        colors[random.nextInt(colors.length)]));
}

```

The *Paint* class is an Android class that is used when drawing the ball onto the screen and contains the information on which color the balls must be drawn with, among other attributes that I left to their default values. To be able to use this class, I need to import its package.

```

<<BallsView imports>>=
import android.graphics.Paint;

```

Then I can use the *Paint* object to draw the ball as a circle on the screen's canvas, which is also an standard Android class.

```

<<BallsView imports>>=
import android.graphics.Canvas;

```

```

<<Ball class>>=
public void draw(Canvas canvas) {
    canvas.drawCircle(position.getX(), position.getY(), radius, paint);
}

```

To draw all the balls in the level, thus, I loop for each ball in the *ArrayList* and call this draw member function.

```

<<draw balls>>=
for(Ball ball: balls) {
    ball.draw(canvas);
}

```

The *Vector2D* class, on the other hand, is a class that I have created in order to have the two values — X and Y — for the position or the direction as a single entity. Besides the constructor, which accepts the coordinates or another *Vector2D* object, this class has a method to “move” the *Vector2D* based on the values of another *Vector2D*.

```

<<Vector2D class>>=
class Vector2D {
    private float x;
    private float y;

    public Vector2D(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public Vector2D(Vector2D other) {
        this(other.getX(), other.getY());
    }

    public float getX() {
        return x;
    }
}

```

```

    }

    public float getY() {
        return y;
    }

    public void setX(float x) {
        this.x = x;
    }

    public void setY(float y) {
        this.y = y;
    }

    public void move(Vector2D other) {
        setX(getX() + other.getX());
        setY(getY() + other.getY());
    }

```

This *move* member function of *Vector2D* comes handy when updating the ball's position with the current direction. This is done in the *update* member function of *Ball* and requires the level's minimum and maximum coordinates to be able to change the current direction when the ball hits the borders.

```

<<Ball class>>=
    public void update(float minX, float minY, float maxX, float maxY) {
        position.move(direction);

        if (position.getX() <= minX) {
            direction.setX(1.0f);
        } else if (position.getX() >= maxX) {
            direction.setX(-1.0f);
        }

        if (position.getY() <= minY) {
            direction.setY(1.0f);
        } else if (position.getY() >= maxY) {
            direction.setY(-1.0f);
        }
    }
}

```

To update all the balls, then, I need to get the view's current height and width while the minimum X and Y are always, by design of the *View* class, 0.

```

<<update balls>>=
for(Ball ball: balls) {
    ball.update(0, 0, getWidth(), getHeight());
}

```

#### 2.4.4 Explosions

Much like the balls, I am going to create a new class that represents an explosion. Unlike the balls, though, the explosions never change their position and instead they grow their radius until they reach a maximum radius, and then shrink into oblivion. The constructor's parameters, though, are the same as the *Ball* class.

```

<<Explosion class>>=
class Explosion {
    private final Vector2D position;
    private final Paint paint;
    private float radius;
}

```

```

private float radiusGrowth;

public Explosion(float x, float y, float radius, int color) {
    this (new Vector2D(x, y), radius, color);
}

public Explosion(Vector2D position, float radius, int color) {
    this.position = new Vector2D(position);
    this.radius = radius;
    this.radiusGrowth = 1.0f;
    this.paint = new Paint();
    this.paint.setColor(color);
}

public float getRadius() {
    return radius;
}

```

There are two constructors for the class *Explosion*. One takes a *Vector2D* as position while the other takes the two values separately. I've done this because when the user taps on the screen, I create the explosion using the coordinates in which the user tapped as given out by the *Event* class.

```

<<BallsView attributes>>=
private ArrayList<Explosion> explosions = new ArrayList<Explosion>();

<<create player explosion>>=
explosions.add(new Explosion(event.getX(), event.getY(), 1, colors[random.nextInt(colors. ←
length])));

```

But when the explosion collides with a ball, then I create the explosion using the ball's position, radius, and color. This new explosion actually replaces the collided ball, thus I have to remove the ball from the *ArrayList* as well and update the status text to show the number of removed balls.

```

<<check collision between explosion and balls>>=
for (ListIterator ballIter = balls.listIterator() ; ballIter.hasNext() ; ) {
    Ball ball = (Ball)ballIter.next();
    if (explosion.collides(ball)) {
        explosionIter.add(new Explosion(ball.getPosition(), ball.getRadius(), ball.getColor ←
        ());
        ballIter.remove();
        updateStatusText();
    }
}

```

Since both the explosion and the balls are actually circles, checking for collision is as easy as checking whether their distance is smaller than the sum of their radius. But, to avoid an unnecessary squared root, instead I check if the squared distance is less than the square of the sum of their radius.

```

<<Explosion class>>=
public boolean collides(Ball ball) {
    float sumRadiusSquared = (this.radius + ball.getRadius());
    sumRadiusSquared *= sumRadiusSquared;

    return this.position.distanceSquared(ball.getPosition()) < sumRadiusSquared;
}

```

*distanceSquared* is defined in the *Vector2D* class as the simple euclidean distance between the two *Vector2D* objects.

```

<<Vector2D class>>=
public float distanceSquared(Vector2D other) {
    return (this.getX() - other.getX()) * (this.getX() - other.getX()) +

```

```

        (this.getY() - other.getY()) * (this.getY() - other.getY());
    }
}

```

Another particularity of both *Ball* and *Explosion* being circles is that drawing the explosions on the view is done exactly the same way as with the balls.

```

<<Explosion class>>=
    public void draw(Canvas canvas) {
        canvas.drawCircle(position.getX(), position.getY(), radius, paint);
    }

```

But, there is a subtle difference in the **order** in which the balls and the explosions must be drawn. In the case of the balls it actually doesn't matter in which order they gets rendered on the screen, but when plotting explosions it is important to draw the last explosion first. The rationale is that later explosions are created by the previous explosions and should be drawn **below** in order to look good, otherwise the new explosion would look like it jumped over the existing. By drawing the last explosion first, I make sure that older explosions overlap younger explosions and the effect looks good on the screen.

The problem with Java is that I can't tell from what item begin iterating except by using explicit iterators. To avoid the ugly syntax of iterators in this case, I wrote a *reversed* static member function that wraps a *ListIterator* within in another iterator whose *next* members calls the original iterator's *previous* member function effectively creating a "reverse iterator".

```

<<BallsView imports>>=
import java.util.List;
import java.util.Iterator;
import java.util.ListIterator;

<<reversed member function>>=
public static<T> Iterable<T> reversed(final List<T> list) {
    return new Iterable<T>() {
        public Iterator<T> iterator() {
            final ListIterator<T> listIter = list.listIterator(list.size());
            return new Iterator<T>() {
                public boolean hasNext() {
                    return listIter.hasPrevious();
                }
                public T next() {
                    return listIter.previous();
                }
                public void remove() {
                    listIter.remove();
                }
            };
        }
    };
}

```

With this "reverse iterator" in place, now I can use a more succinct syntax to draw the explosions.

```

<<draw explosions>>=
for (Explosion explosion: reversed(explosions)) {
    explosion.draw(canvas);
}

```

The only thing remaining for the *Explosion* class is to update itself. In the case of explosions, as already stated, they should grow until they reach a maximum radius, passed to their *update* member function as parameter. Once they reach this maximum, the explosions shall shrink until their radius is 0, in which case the explosion is considered to be finished and no longer updates itself.

```

<<Explosion class>>=
    public void update(float maximumRadius) {
        if (radius > 0.0f) {

```

```

        radius += radiusGrowth;
        if (radius >= maximumRadius) {
            radiusGrowth = -1;
        }
    }
}

```

But when updating the explosions, the game has to take into account that they could be colliding with any of the bouncing balls on the screen, creating more explosions, and that explosions already finished must be deleted. This translates in adding or removing elements in the explosions' *ArrayList*, hence I can't use the *for* syntax and must use iterators because I can't add or delete elements in the *ArrayList* directly while iterating. I must use the iterator to add or delete.

```

<<BallsView constants>>=
private static final float EXPLOSION_MAXIMUM_RADIUS = 30;

```

```

<<update explosions>>=
for(ListIterator explosionIter = explosions.listIterator() ; explosionIter.hasNext() ; ) {
    Explosion explosion = (Explosion)explosionIter.next();
    explosion.update(EXPLOSION_MAXIMUM_RADIUS);
    if (explosion.getRadius() > 0.0f) {
        <<check collision between explosion and balls>>
    } else {
        explosionIter.remove();
    }
}

```

## 2.4.5 Updating the Game State

In traditional consoles and PC games, I usually have a single main loop in where I update the current game's state and make sure that everything goes smoothly. In Android I can't have that because who is running the activity, and thus has the main loop, is Android not me. That means that in order to keep ticking the game, I need to use a different approach.

What I do is have a *Handler* derived class that sends messages to itself periodically and, in its message handler, calls *BallsView update* member function as well as invalidating the screen's contents to force Android to redraw the view.

```

<<BallsView imports>>=
import android.os.Handler;
import android.os.Message;

```

```

<<Refresh handler>>=
class RefreshHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        BallsView.this.update();
        BallsView.this.invalidate();
    }

    public void sleep(long delayMs) {
        this.removeMessages(0);
        sendMessageDelayed(obtainMessage(0), delayMs);
    }
}

```

```

<<BallsView attributes>>=
private final RefreshHandler refreshHandler = new RefreshHandler();

```

*BallsView update* member function is the function that actually calls every other object's *update* member function to keep the game running and schedules a new call to itself using *refreshHandler*. This only must happen while the level is playing, which is the same as saying that while *nextLevelTextView* remains invisible.

```
<<BallsView constants>>=
private static final long GAME_SPEED = 50;

<<update member function>>=
public void update() {
    if (nextLevelTextView.getVisibility() == View.INVISIBLE) {
        <<update balls>>
        boolean hadExplosions = !explosions.isEmpty();
        <<update explosions>>
        <<check end of level>>
        refreshHandler.sleep(GAME_SPEED);
    }
}
```

The update is also responsible to check whether the level is finished or not. The simplest way to check that is detecting if the level had to remove all the ongoing explosions. As we know that the player can only start a single explosion per level, if the game went from an state that has explosions (i.e., the player can't add more explosions) to an state without any then there's nothing more to do other than to check whether to advance to the next level.

The game only advances to the next level if the number of balls removed is equal or greater than the level's target. If this is the case, then I increment the current level, trying not to go beyond the maximum number of levels, and ask the player to tap to start again. Otherwise, it remains on the same level and prompts the user to tap to try again. In both cases, the status text is set to *INVISIBLE*, the text asking to tap is set to *VISIBLE* and all the balls are removed.

```
<<BallsView imports>>=
import android.content.res.Resources;

<<check end of level>>=
Resources resources = getContext().getResources();
CharSequence text = "";
if (hadExplosions && explosions.isEmpty()) {
    if (countRemovedBalls() >= levels[currentLevel].target) {
        ++currentLevel;
        if (currentLevel > levels.length - 1) {
            currentLevel = levels.length - 1;
        }
        text = resources.getText(R.string.next_level);
    } else {
        text = resources.getText(R.string.try_again);
    }
    balls.clear();
    statusTextView.setVisibility(View.INVISIBLE);
    nextLevelTextView.setText(text);
    nextLevelTextView.setVisibility(View.VISIBLE);
}
```

See how instead of hardcoding the texts to display to the user, I am using the strings declared in the strings file (see Section 2.5).

```
<<try again string>>=
<string name="try_again">Oops! Try Again</string>
```

To count the number of removed balls I subtract the remaining number of balls from the level's initial number of balls.

```
<<count removed balls member>>=
public int countRemovedBalls() {
    return levels[currentLevel].balls - balls.size();
}
```



### 2.4.6 Pushing the Pixels on the Screen

To actually see any of the previous classes gaily dancing on the device as pixies, I need to instruct the view to actually draw them. Again, differing from other system, in Android I don't have a main loop in which I can tell the application to flush the contents to the screen. The view must wait until Android asks for the it to draw itself by calling its *onDraw* member function and passing the *Canvas* to draw to when the screen is invalidated by *refreshHandler*. I need to override this function and draw each ball and explosion from the respective *ArrayList*.

```
<<draw member function>>=
@Override
public void onDraw(Canvas canvas) {
    <<draw balls>>
    <<draw explosions>>
}
```

Notice how I don't have to draw the contents of any of the *TextView* objects. Those are defined in the layout file, so Android takes this responsibility. Less work for me.

### 2.4.7 Tap, Tap, Tapping

The only remaining glue for this game to work is input handling. Being a touch based game, I need to know when the user taps on the screen. Android calls the *onTouchEvent* every time there is anything related to touch: when the user puts the greasy finger on the screen, when puts it up, when scrolling, et cetera. In the case of *Balls!*, the only event I am interested on is *ACTION\_UP* fired when the user presses the screen no more.

If the game is waiting for the user to start the game with a first tap (i.e., *nextLevelTextView* is visible) the game hides that object, makes the *statusTextView* visible, updates its text and adds the required balls according to the current level. Otherwise, if there is no explosion already created, add a new explosion at the position where the touch event occurred. In all cases it calls the *update* method to start the update loop.

```
<<BallsView imports>>=
import android.view.MotionEvent;

<<touch event handler>>=
@Override
public boolean onTouchEvent(MotionEvent event) {
    if (event.getAction() == MotionEvent.ACTION_UP) {
        if (nextLevelTextView.getVisibility() == View.VISIBLE) {
            nextLevelTextView.setVisibility(View.INVISIBLE);
            <<create balls>>
            updateStatusText();
            statusTextView.setVisibility(View.VISIBLE);
            update();
        } else {
            if (explosions.isEmpty()) {
                <<create player explosion>>
            }
        }
    }
    return true;
}
```

The *updateStatusText* member function sets the text for the *statusTextView* to show the current number of removed balls and the level's target.

```
<<update status text>>=
public void updateStatusText() {
    statusTextView.setText(String.format("%d/%d", countRemovedBalls(), levels[currentLevel ←
    ].target));
}
```

### 2.4.8 BallsView.java

All the previous subsection can be placed inside the *BallsView* class as follows.

```
<<BallsView.java>>=
//
// Balls! -- A simple game with balls.
// Copyright 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
package com.geishastudios.balls;

import android.view.View;
<<BallsView imports>>

public class BallsView extends View {
    <<BallsView constants>>

    <<Level class>>

    <<Vector2D class>>

    <<Ball class>>

    <<Explosion class>>

    <<Refresh handler>>

    <<BallsView attributes>>

    <<BallsView constructors>>

    <<reversed member function>>

    <<setting the text views>>

    <<draw member function>>

    <<update member function>>

    <<touch event handler>>

    <<count removed balls member>>

    <<update status text>>
}
```

## 2.5 Strings File

Usually, Android's applications keep the strings to display to the user apart from the source code. This is done to easier localization (l10n) and internationalization (i18n) of the application. This means that by keeping the strings in a different file, we could have different files for each language and the application would show the correct string depending on the user's language.

In the case of this game, I only use a single file. This file is in `res/values` and is called `strings.xml`. This file is an XML file as well.

```
<<strings.xml>>=
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <<application name string>>
```

```
<<press tab to start string>>  
<<try again string>>  
</resources>
```

## A License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=  
// This program is free software; you can redistribute it and/or modify  
// it under the terms of the GNU General Public License version 2.0 as  
// published by the Free Software Foundation.  
//  
// This program is distributed in the hope that it will be useful,  
// but WITHOUT ANY WARRANTY; without even the implied warranty of  
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
// GNU General Public License for more details.  
//  
// You should have received a copy of the GNU General Public License  
// along with this program; if not, write to the Free Software  
// Foundation, Inc., 50 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```