

iTouchBalls

COLLABORATORS

	<i>TITLE :</i>		
	iTouchBalls		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jordi Fita	May 31, 2011	

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.	jfit
6cc909c0b61d	2011-03-07	Added the comments section.	jfit
63a438521347	2011-03-07	Added a missing snippet name.	jfit
158b4beced10	2011-03-07	Reworded some of the paragraphs to easier understanding (I hope)	jfit
e73be3b3a6c9	2011-03-07	Fixed a couple of source tags.	jfit
882f083478ee	2011-03-06	Added the status text at the bottom.	jfit
796441ded6d8	2011-03-06	Added the drawing of the next level status.	jfit
13761768abb1	2011-03-06	Added the check whether the level is finished.	jfit
307a1a94ef1f	2011-03-05	Using 15FPS instead of 30.	jfit
dd168af98daf	2011-03-04	Added the explosion on tap.	jfit
1e548eb394bb	2011-03-04	Added the collision function between balls and explosions.	jfit

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
130cee844f04	2011-03-03	Added the Explosion class.	jfita
1a569a15a4ab	2011-02-26	Added the creation of balls on tap.	jfita
4f9c425a068e	2011-02-21	Added the Level class and the initialization of the levels NSArray.	jfita
df31631cf0a4	2011-02-21	Added the drawing of balls.	jfita
a54aa5df6113	2011-02-19	The array of balls is now NSMutableArray instead.	jfita
b6e068d41ab4	2011-02-19	Added the balls array and removed the starting variable.	jfita
4788a8fb383c	2011-02-19	Added the Ball class.	jfita
f856f7fbcbbd	2011-02-19	Added the 'started' property to BallsView.	jfita
7a82ef52fc05	2011-02-19	Added the main loop.	jfita
25bee09c8a1f	2011-02-19	Fixed the section level for 'plist file'.	jfita
da426a4818c0	2011-02-19	Fixed the section level of 'Main' section.	jfita
7477ba944a7f	2011-02-19	Added the empty BallsView. Changed the window's frame to use bounds instead.	jfita
02627d79c046	2011-02-19	Added the ITouchBall class inside the 'main' section, as it is really "an extension" of it.	jfita
c6601b32b5c1	2011-02-19	Fixed mistakes in plist file for iTouchBalls.	jfita
08a069444f82	2011-02-19	Fixed a typo in iTouchBalls' main.m	jfita
79730c7a3282	2011-02-19	Added the main source code to iTouchBalls.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
d5378c1f4ba5	2011-02-19	Added the prefix header to iTouchBalls.	jfita
25ec21dcf298	2011-02-19	Added the license text to iTouchBalls.	jfita
e29bff26ecb	2011-02-18	Added the initial iTouchBall projects and the plist file.	jfita

Contents

1	Introduction	1
2	Download	1
3	Code	1
3.1	The View	1
3.2	Main Loop	2
3.3	Status Text	4
3.4	Levels	4
3.5	Balls	6
3.6	Explosions	9
3.7	Tap, Tap, Tapping	13
A	Main	14
B	Prefix Header	16
C	Plist File	16
D	License	17

1 Introduction

iTouchBalls is an **iOS** port of **Balls!**, a very simple puzzle game in which the player must remove a specific number of bouncing balls 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 iOS application, *iTouchBalls* is written in Objective-C.

2 Download

There is no precompiled version of *iTouchBalls* available because I highly doubt that Apple would allow this game in his **App Store**. However, the source code, extracted with **atangle**, and an **Xcode** project file are available at the following URL:

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

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

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

3 Code

3.1 The View

Even though Apple recommends to separate the application in three different responsibilities using a Model-View-Controller (MVC) pattern, I've decided to put everything in a single `UIView` derived class which I named `BallsView`.

```
<<BallsView.h>>=
//
// iTouchBalls - A simple game with balls.
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
#import <UIKit/UIKit.h>

@interface BallsView: UIView {
    <<BallsView interface>>
}

@end
```

```
<<BallsView.m>>=
//
// iTouchBalls - A simple game with balls.
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
<<BallsView includes>>
#import "BallsView.h"
#import "Ball.h"
#import "Explosion.h"
#import "Level.h"

@implementation BallsView

<<view constructor>>
```

```

<<update function>>

<<draw function>>

<<tap function>>

- (void)dealloc {
    <<release main loop>>
    <<release BallsView interface>>
    [super dealloc];
}

@end

```

This class must be initialized by calling the `UIView` inherited `initWithFrame` function and passing the frame with the view's size.

```

<<create view>>=
BallsView *view = [[BallsView alloc] initWithFrame:frame];

```

This method, besides calling its parent's method and initialize the object's interface, sets the view's background color as well as the application's main loop which will update the game's logic and force a redraw of the view.

```

<<view constructor>>=
- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        <<init BallsView interface>>
        self.backgroundColor = [UIColor colorWithRed:42.0f / 255.0f green:84.0f / 255.0f ←
            blue:128.0f / 255.0f alpha:1.0f];
        <<set up main loop>>
    }
    return self;
}

```

Once the view is created, I can set it to the application's window to make the view visible. The function that I am using to set the view, `setContentView`, is not documented and, in fact, XCode will complain that `UIWindow` might not respond to this message. The method exists and I use it because I remember using it from regular OS X development. Unfortunately, I wasn't able to set the view using any other method.

```

<<set view in window>>=
[window setContentView:view];

```

3.2 Main Loop

Differing from traditional imperative applications, an iOS application does not have the common even loop in which the application can check for incoming events and act upon them. This is all handled by the `UIApplicationMain` function, among other initialization and set up routines. Unfortunately, for games, this means that I no longer have the option to add the calls to the methods that update the game's internal logic each frame in that loop. Instead, I need to find a way to call my update function periodically. This is where `NSTimer` comes to help me.

With `NSTimer` I can schedule a call to a function of my choice at some point in the future and this call will be repeated until the timer gets invalidated. Given that I want to update the game's logic at an speed of 15 frames per second, I need to call the update function every 66 milliseconds ($1000 / 15$).

```

<<set up main loop>>=
mainLoop =
    [NSTimer scheduledTimerWithTimeInterval:0.066f

```

```

        target:self
        selector:@selector(update)
        userInfo:nil
        repeats:YES];

```

The `mainLoop` variable is defined in the `BallsView` class and released when the class is deallocated.

```

<<BallsView interface>>=
NSTimer *mainLoop;

```

```

<<release main loop>>=
[mainLoop release];

```

The function referenced in the `selector` parameter is the function that will be called repeatedly every 0.066 seconds. This function's responsibility is to update the balls and the explosions in the game as well as to check whether the game is done. Once these are updated, I trigger a redraw by telling iOS that the view needs to display again.

```

<<update function>>=
- (void)update {
    <<update balls>>
    <<check if there is any explosion>>
    <<update explosions>>
    <<check whether level is over>>
    [self setNeedsDisplay];
}

```

The drawing is performed in `drawRect: .` Here I need to get the current graphic's context and then simply draw the balls, the explosions, and the status text onto this context, if there is any ball on the level. If there are no balls yet, I need to tell the player to tap the screen to start the level.

```

<<draw function>>=
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    if ([balls count] == 0) {
        <<draw next level label>>
    } else {
        <<draw balls>>
        <<draw explosions>>
        <<draw status>>
    }
    CGContextStrokePath(context);
}

```

To draw the label, I am going to use `'NSString'` `drawAtPoint:withFont` which prints the text at the given point on the current context — the one I've got when called `UIGraphicsGetCurrentContext` — with the specified font. As I want this text to be drawn at the view's center, first I need to compute the position where to draw the text. Fortunately, `NSString` also has the `sizeWithFont:` function that returns the rendered string's size if it were drawn using the passed font. Knowing the text's size as well as the view's size, the position on there to start drawing is half the view's size less half the text size, in both vertical and horizontal.

```

<<draw next level label>>=
CGRect viewRect = [self bounds];
UIFont *font = [UIFont boldSystemFontOfSize:20.0];
CGSize size = [nextLevelLabel sizeWithFont:font];
CGPoint pos = CGPointMake(viewRect.origin.x + (viewRect.size.width - size.width) / 2,
                           viewRect.origin.y + (viewRect.size.height - size.height) / 2);
[[UIColor whiteColor] set];
[nextLevelLabel drawAtPoint:pos withFont:font];

```

The next level label is just a string which prompts the user to tap on the screen to start the level.

```
<<BallsView interface>>=
NSString *nextLevelLabel;
```

```
<<init BallsView interface>>=
nextLevelLabel = @"Tap to Start Next Level";
```

3.3 Status Text

The status text is an string with the number of balls removed from the level and the target balls to remove. I need to get the current level and the remaining balls on the screen to know the number of removed balls, that I can get with a simple subtraction.

```
<<draw status>>=
Level *level = [levels objectAtIndex:currentLevel];
int removedBalls = level.balls - [balls count];
NSString *status = [NSString stringWithFormat:@"%d/%d", removedBalls, level.target];
```

This string needs to be drawn at the screen's bottom right corner. Again, I am using `NSString sizeWithFont` : to know the area used by the text with the intended font but this time I have to subtract the whole text's size from the view's size to render the text at the bottom's right corner.

```
<<draw status>>=
CGRect viewRect = [self bounds];
UIFont *font = [UIFont systemFontOfSize:15.0];
CGSize size = [status sizeWithFont:font];
CGPoint pos = CGPointMake(viewRect.origin.x + viewRect.size.width - size.width,
                           viewRect.origin.y + viewRect.size.height - size.height);
[[UIColor whiteColor] set];
[status drawAtPoint:pos withFont:font];
```

3.4 Levels

The levels only requires two pieces of information: the number of balls bouncing on the screen and the minimum number of balls to remove, known as target. These two never change during the lifetime of a level and thus can be set read-only.

```
<<Level.h>>=
//
// iTouchBalls - A simple game with balls
// Copyright (c) 2011 Jordi Fita
//
<<license>>
//
#import <Foundation/Foundation.h>

@interface Level: NSObject {
    unsigned int balls;
    unsigned int target;
}

@property (readonly) unsigned int balls;
@property (readonly) unsigned int target;
```

These two values are initialized in the `Level's initWithBalls:target:` method.

```
<<Level.h>>=
- (id)initWithBalls:(unsigned int)balls target:(unsigned int)target;
```

```

<<Level.m>>=
//
// iTouchBalls - A simple game with balls.
// Copyright (c) 2011 Jordi Fita
//
<<license>>
//
#import "Level.h"

@implementation Level

@synthesize balls;
@synthesize target;

- (id)initWithBalls:(unsigned int)balls_ target:(unsigned int)target_ {
    self = [super init];
    if (self) {
        balls = balls_;
        target = target_;
    }

    return self;
}

```

The levels are initialized by the BallsView when created and they are stored in an NSArray object. I also keep a variable to know in which level the game is currently. This variable is initially set to the first level, which is 0.

```

<<BallsView interface>>=
NSArray *levels;
unsigned int currentLevel;

```

For convenience, I am going to add a class method to Level that creates the level and calls autorelease on it, thus freeing me to have to keep the references and releasing them when adding them to the array.

```

<<Level.h>>=
+ (id)levelWithBalls:(unsigned int)balls target:(unsigned int)target;

@end

```

```

<<Level.m>>=
+ (id)levelWithBalls:(unsigned int)balls target:(unsigned int)target {
    Level *level = [[Level alloc] initWithBalls:balls target:target];
    return [level autorelease];
}

@end

```

With this method in place, now I can create the levels thus:

```

<<init BallsView interface>>=
levels = [[NSArray alloc] initWithObjects:
    [Level levelWithBalls:4 target:1],
    [Level levelWithBalls:4 target:2],
    [Level levelWithBalls:8 target:4],
    [Level levelWithBalls:15 target:7],
    [Level levelWithBalls:25 target:12],
    [Level levelWithBalls:30 target:17],
    [Level levelWithBalls:30 target:20],
    [Level levelWithBalls:30 target:25],
    [Level levelWithBalls:30 target:28],
    [Level levelWithBalls:30 target:30],

```

```
    nil];  
currentLevel = 0;
```

The array must be released when the BallsView is done.

```
<<release BallsViews interface>>=  
[levels release];
```

3.5 Balls

The balls in this game are just circles that bounce around the screen. Thus, they need a position, a direction, a radius and a color. But, to make it easier to draw onto a Quartz context, I am going to store the balls as the rectangle that encloses the circle, the direction it is moving, and its color.

```
<<Ball.h>>=  
//  
// iTouchBalls - A simple game with balls.  
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>  
//  
<<license>>  
//  
#import <UIKit/UIKit.h>  
  
@interface Ball: NSObject {  
    UIColor *color;  
    CGPoint dir;  
    CGRect rect;  
}  
  
<<Ball constructor declaration>>  
<<Ball draw declaration>>  
<<Ball update declaration>>  
  
@property (nonatomic, retain) UIColor *color;  
@property (nonatomic) CGPoint dir;  
@property (nonatomic) CGRect rect;  
<<Ball radius property declaration>>  
<<Ball position property declaration>>  
  
@end
```

```
<<Ball.m>>=  
//  
// iTouchBalls - A simple game with balls.  
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>  
//  
<<license>>  
//  
#import "Ball.h"  
  
@implementation Ball  
  
@synthesize color;  
@synthesize dir;  
@synthesize rect;  
  
<<Ball constructor>>  
  
<<Ball draw>>
```

```

<<Ball update>>

<<Ball radius property>>

<<Ball position property>>

- (void)dealloc {
    [color release];
    [super dealloc];
}

@end

```

Ball's constructor expects a point, a radius and a color but then constructs the rectangle centered at the given point. The initial direction is towards the screen's top-left corner.

```

<<Ball constructor declaration>>=
- (id)initWithPosition:(CGPoint)pos radius:(CGFloat)radius color:(UIColor *)color;

<<Ball constructor>>=
- (id)initWithPosition:(CGPoint)pos radius:(CGFloat)radius color:(UIColor *)ballColor {
    self = [super init];
    if (self) {
        [ballColor retain];
        self.color = ballColor;
        self.dir = CGPointMake(-1.0f, -1.0f);
        self.rect = CGRectMake(pos.x - radius, pos.y - radius, radius * 2, radius * 2);
    }
    return self;
}

```

I do this because then drawing the ball to a context is just a matter to pass the appropriate attributes to the functions.

```

<<Ball draw declaration>>=
- (void)drawToContext:(CGContextRef) context;

<<Ball draw>>=
- (void)drawToContext:(CGContextRef) context {
    CGContextSetFillColorWithColor(context, self.color.CGColor);
    CGContextFillEllipseInRect(context, self.rect);
}

```

Thus drawing all the balls is a matter of calling this function with the context to draw on.

```

<<draw balls>>=
for (Ball *ball in balls) {
    [ball drawToContext:context];
}

```

Although this approach makes updating the balls a little more complicated because now, besides updating the current position according to the direction, I have to find the rectangle's center position in order to know whether to change the direction when the ball must bounce around the screen's limits, which are passed as a parameter when updating.

Given that I'll use the balls position later to know whether the balls collides with an explosion, I think it is better to put this center position in a position property of Ball. This property returns the Ball's enclosing rectangle's origin point but with an offset equal to the ball's radius.

```

<<Ball position property declaration>>=
@property (nonatomic, readonly) CGPoint position;

```

```
<<Ball position property>>=
- (CGPoint)position {
    return CGPointMake(self.rect.origin.x + self.radius,
                       self.rect.origin.y + self.radius);
}
```

The ball's radius is also a property, for the same reason as the position, and can also be extracted from the ball's rectangle, knowing that both the rectangle's width and height — they are the same value — are the ball's diameter, the radius is half either of these values.

```
<<Explosion radius property declaration>>=
@property (nonatomic, readonly) CGFloat radius;
```

```
<<Explosion radius property>>=
- (CGFloat)radius {
    return self.rect.size.width / 2;
}
```

With these two properties, it is now easy to write the function that updates the ball's position.

```
<<Ball update declaration>>=
- (void)updateWithBounds:(CGRect)bounds;
```

```
<<Ball update>>=
- (void)updateWithBounds:(CGRect)bounds {
    self.rect = CGRectMake(
        self.rect.origin.x + self.dir.x,
        self.rect.origin.y + self.dir.y,
        self.rect.size.width, self.rect.size.height);

    CGPoint center = self.position;
    if (center.x < bounds.origin.x) {
        self.dir = CGPointMake(1.0f, self.dir.y);
    } else if (center.x >= bounds.origin.x + bounds.size.width) {
        self.dir = CGPointMake(-1.0f, self.dir.y);
    }
    if (center.y < bounds.origin.y) {
        self.dir = CGPointMake(self.dir.x, 1.0f);
    } else if (center.y >= bounds.origin.y + bounds.size.height) {
        self.dir = CGPointMake(self.dir.x, -1.0f);
    }
}
```

Calling the update function is as easy as drawing, but instead of passing the context, I need to pass the view's bounds rectangle to keep the balls within the view.

```
<<update balls>>=
CGRect bounds = [self bounds];
for (Ball *ball in balls) {
    [ball updateWithBounds:bounds];
}
```

balls, both when drawing and when updating, is an object of type NSMutableArray owned by BallsViews and thus is created when initializing BallsView.

```
<<BallsView interface>>=
NSMutableArray *balls;
```

```
<<init BallsView interface>>=
balls = [[NSMutableArray alloc] init];
```

Of course, since `BallsView` has ownership of this object, it is also responsible of releasing it.

```
<<release BallsView interface>>=  
[balls release];
```

3.6 Explosions

Like the balls, the explosions in `iTouchBalls` are simple circles. But, unlike the balls, instead of moving around the screen they change their radius in such a way to mimic an explosion's blast. Like I did for the balls, I am going to store the explosions as the rectangle that encloses the explosion and its color. The explosion has also a *growth rate* which is the speed in which the explosion grows or shrinks.

```
<<Explosion.h>>=  
//  
// iTouchBalls - A simple game with balls.  
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>  
//  
<<license>>  
//  
#import <UIKit/UIKit.h>  
  
@class Ball;  
  
@interface Explosion: NSObject {  
    UIColor *color;  
    CGFloat growthRate;  
    CGRect rect;  
};  
  
<<Explosion constructor declaration>>  
<<Explosion draw declaration>>  
<<Explosion update declaration>>  
<<Explosion collides declaration>>  
  
@property (nonatomic, retain) UIColor *color;  
@property (nonatomic) CGFloat growthRate;  
@property (nonatomic) CGRect rect;  
<<Explosion radius property declaration>>  
  
@end
```

```
<<Explosion.m>>=  
//  
// iTouchBalls - A simple game with balls.  
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>  
//  
<<license>>  
//  
#import "Explosion.h"  
#import "Ball.h"  
  
@implementation Explosion  
  
@synthesize color;  
@synthesize growthRate;  
@synthesize rect;  
  
<<Explosion constructor>>  
  
<<Explosion draw>>
```

```

<<Explosion update>>

<<Explosion collides>>

<<Explosion radius property>>

- (void)dealloc {
    [color release];
    [super dealloc];
}

@end

```

Explosion's constructor expects a position, a radius, and a color but then constructs the rectangle centered at the specified point. The initial growth ratio is set to 1 pixel per each update.

```

<<Explosion constructor declaration>>=
- (id)initWithPosition:(CGPoint)pos radius:(CGFloat)radius color:(UIColor *)color;

```

```

<<Explosion constructor>>=
- (id)initWithPosition:(CGPoint)pos radius:(CGFloat)radius color:(UIColor *)explosionColor ←
{
    self = [super init];
    if (self) {
        [explosionColor retain];
        self.color = explosionColor;
        self.growthRate = 1.0f;
        self.rect = CGRectMake(pos.x - radius, pos.y - radius, radius * 2, radius * 2);
    }
    return self;
}

```

This rectangle centered around the explosion renders the drawing of said explosion dead simple, just like I did with the balls.

```

<<Explosion draw declaration>>=
- (void)drawToContext:(CGContextRef) context;

```

```

<<Explosion draw>>=
- (void)drawToContext:(CGContextRef) context {
    CGContextSetFillColorWithColor(context, self.color.CGColor);
    CGContextFillEllipseInRect(context, self.rect);
}

```

Calling this function for all explosions and passing the view's context draws them all on the screen.

```

<<draw explosions>>=
for (Explosion *explosion in explosions) {
    [explosion drawToContext:context];
}

```

But updating the rectangle is now a little more involved than just updating the radius property. Now I have to grow or shrink the explosion's rectangle around the center. Given that the growth rate is exactly how many pixels the radius grow each update, I only have to move the rectangle's position **away** from the center by that amount while, at the same time, I make the rectangle bigger by **double** that amount, because the rectangle's size represents the diameter, not the radius. Of course, if the growth is negative (i.e., shrinking) then the operation is reversed, but this happens automatically because the growth rate's variable is negative.

Another thing to take into account when updating the explosions is that they grow until a certain limit. This limit is passed as a parameter to the update function, but this limit is the maximum radius, which is half the rectangle's width or height. When this limit is reached, then the explosion starts to shrink until the ball's diagonal is 0.

```
<<Explosion update declaration>>=
- (void)updateWithMaxRadius:(CGFloat)radius;
```

```
<<Explosion update>>=
- (void)updateWithMaxRadius:(CGFloat)maxRadius {
    if (self.radius > 0) {
        self.rect = CGRectMake(
            self.rect.origin.x - self.growthRate,
            self.rect.origin.y - self.growthRate,
            self.rect.size.width + self.growthRate * 2,
            self.rect.size.height + self.growthRate * 2);

        if (self.radius / 2 > maxRadius) {
            self.growthRate = -1;
        }
    }
}
```

To make the code a little easier to understand, I've added a new property to `Explosion` which hides the details of how I need to extract the radius from the explosion's rectangle.

```
<<Ball radius property declaration>>=
@property (nonatomic, readonly) CGFloat radius;
```

```
<<Ball radius property>>=
- (CGFloat)radius {
    return self.rect.size.width / 2;
}
```

Although the idea behind `Explosion`'s `updateWithMaxRadius:` is the same as the ball's `updateWithBounds:`, traversing and updating the explosions is not as straightforward as with the balls. The main difference is that besides calling the update function, I have to check whether the explosions collide with a ball or not. I also have to make sure that explosions no longer valid (i.e., its radius reached 0) get removed from the game.

```
<<update explosions>>=
for (int explosionIndex = 0 ; explosionIndex < [explosions count] ; ++explosionIndex) {
    Explosion *explosion = [explosions objectAtIndex:explosionIndex];
    [explosion updateWithMaxRadius:30.0f];
    <<check whether to remove explosion>>
    <<otherwise check whether explosion collides with balls>>
}
```

The check whether the explosion must be removed, I need to know its radius. If the radius reached 0, then the explosion is no longer valid and must be removed from the array. I also need to subtract one from the explosion's index or I would skip the next explosion, because removing the explosion moves all posterior explosions one position to the beginning of the array.

```
<<check whether to remove explosion>>=
if (explosion.radius <= 0) {
    [explosions removeObjectAtIndex:explosionIndex];
    --explosionIndex;
}
```

If the explosion is still active (i.e., its radius is greater than 0) then I need to check whether collides with any of the balls on the screen. If the explosion is colliding with a ball, that ball is replaced with a new explosion at the same position and with the same color as the colliding ball. The new explosion is added at the array's beginning in order to be drawn *under* older explosions. This is because I am drawing the explosions from the first index to the last. If the drawing for-loop was done in reverse, then putting the explosion at the array's end would be enough, and it could even improve the application's performance.

```

<<otherwise check whether explosion collides with balls>>=
else {
    for (int ballIndex = 0 ; ballIndex < [balls count] ; ++ballIndex) {
        Ball *ball = [balls objectAtIndex:ballIndex];
        if ([explosion collidesWithBall:ball]) {
            Explosion *newExplosion =
                [[Explosion alloc] initWithPosition:ball.position
                 radius:ball.radius
                 color:ball.color];
            [explosions insertObject:newExplosion atIndex:0];
            [newExplosion release];
            [balls removeObjectAtIndex:ballIndex];
            --ballIndex;
        }
    }
}

```

Knowing that both the explosion and the ball are circles, I can compute the euclidean distance between the two to check whether they are colliding or not. If the distance is less than the sum of their radius, then the explosion and the ball are colliding. To avoid the square root in computing the euclidean distance, instead of the actual distance, which I do not require, I am going to check whether the *squared* distance is less than the *squared* sum of their radii.

```

<<Explosion collides declaration>>=
- (BOOL)collidesWithBall:(Ball *)ball;

```

```

<<Explosion collides>>=
- (BOOL)collidesWithBall:(Ball *)ball {
    CGFloat maxDistance = (self.radius + ball.radius) * (self.radius + ball.radius);
    CGPoint center = CGPointMake(self.rect.origin.x + self.radius,
                                 self.rect.origin.y + self.radius);
    CGPoint ballCenter = ball.position;
    return ((center.x - ballCenter.x) * (center.x - ballCenter.x) +
            (center.y - ballCenter.y) * (center.y - ballCenter.y)) < maxDistance;
}

```

The only last thing to check for is whether the level is done or not. The level is done when the last explosion is fired and destroyed. This happens when there was any explosion before the last update and then there is not more explosion left. Since the player can only start a single explosion per level, if there is no longer an explosion available, then I can assume there is nothing else to do.

```

<<check if there is any explosion>>=
BOOL anyExplosionLeft = [explosions count] > 0;

```

```

<<check whether level is over>>=
if (anyExplosionLeft && [explosions count] == 0) {

```

To end the level, the only thing I have to do is remove all the balls from their array. But before that, I need to check if we reached the level's target amount of balls to remove. With the number of balls still left and the number of balls that there were initially, I can readily find out whether the level is completed or not. If it is, then move to the next level, if there are more levels.

```

<<check whether level is over>>=
Level *level = [levels objectAtIndex:currentLevel];
if (level.balls - [balls count] >= level.target) {
    nextLevelLabel = @"Tap to Start Next Level";
    if (currentLevel < [levels count] - 1) {
        ++currentLevel;
    }
} else {
    nextLevelLabel = @"Oops. Try Again!";
}

```

```
[balls removeAllObjects];
}
```

explosions, like balls above, is an object of class NSMutableArray owned by BallsView and created with it.

```
<<BallsView interface>>=
NSMutableArray *explosions;
```

```
<<init BallsView interface>>=
explosions = [[NSMutableArray alloc] init];
```

As BallsView has ownership of this object, it is also responsible to release them.

```
<<release BallsView interface>>=
[explosions release];
```

3.7 Tap, Tap, Tapping

Now that I have most of the pieces in place, I need to be able to start the game by tapping on the screen.

When starting, the game waits for the player to tap the screen before creating the balls for the current level. I know when the player tapped the screen because BallsView receives the touchesEnded:withEvent message from the underlying system. I've chosen touchesEnded:withEvent instead of touchesBegan:withEvent because I am going to use the same message to create the explosions and I want a way to rectify, as it were, the explosion's initial position if I did choose the wrong spot.

```
<<tap function>>=
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    <<create balls if level has not already started>>
    <<otherwise create explosion if no explosion exists>>
}
```

In this message I thus need to know whether the level has already started. I can easily know that by looking at the number of elements inside the balls array. If this array is empty, the level hasn't started yet and I must start the game by creating as many balls as dictated by the current level.

```
<<create balls if level has not already started>>=
if ([balls count] == 0) {
    <<get current level>>
    <<create levels balls>>
}
```

Given that the view's currentLevel property is always within the array's bounds, the current level object can be retrieved from the array using this property as the index.

```
<<get current level>>=
Level *level = [levels objectAtIndex:currentLevel];
```

Using this object, I can get the number of balls to create for the current level using a for loop. The balls are created at a random position inside the view's bounds and each has a random color.

```
<<create levels balls>>=
NSArray *colors = [[NSArray alloc] initWithObjects:
    [UIColor redColor],
    [UIColor greenColor],
    [UIColor blueColor],
    [UIColor cyanColor],
    [UIColor yellowColor],
    [UIColor magentaColor],
```

```

    nil];
CGRect bounds = [self bounds];
for (unsigned int currentBall = 0 ; currentBall < level.balls ; ++currentBall) {
    CGPoint position =
        CGPointMake(bounds.origin.x + arc4random() % (int)bounds.size.width,
                    bounds.origin.y + arc4random() % (int)bounds.size.height);
    UIColor *color = [colors objectAtIndex:(arc4random() % [colors count])];
    Ball *ball = [[Ball alloc] initWithPosition:position
                                                radius:4.0f
                                                color:color];

    [balls addObject:ball];
    [ball release];
}
[colors release];

```

The `arc4random()` is a function that returns a pseudo-random number between 0 and $2^{32}-1$ and doesn't require a seed value, unlike `rand()`. Thus, it is more convenient in this case, but I need to include the `stdlib` header where it is declared.

```

<<BallsView includes>>=
#include <stdlib.h>

```

The explosion is easier to create than the balls. The player can only create a single explosion, hence I need to check whether there is already an explosion in the level or not. If there is none, then I create a 1 pixel explosion at the position where the player tapped the screen. I can't create a 0 pixel explosion because when an explosion reaches 0 it gets removed from the level.

```

<<otherwise create explosion if no explosion exists>>=
else if ([explosions count] == 0) {
    UITouch *touch = [touches anyObject];
    Explosion *newExplosion =
        [[Explosion alloc] initWithPosition:[touch locationInView:self]
                                         radius: 1.0f
                                         color: [UIColor redColor]];

    [explosions addObject:newExplosion];
    [newExplosion release];
}

```

A Main

The main file for an iOS application is really simple. I only have to call `UIApplicationMain` passing the arguments from the command line and telling which is the application class. As with most Objective-C applications, the main function is wrapped inside an `NSAutoreleasePool` that manages the application's memory allocation.

```

<<main.m>>=
//
// iTouchBalls - A simple game with balls.
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, @"iTouchBalls", @"iTouchBalls");
    [pool release];

    return retVal;
}

```

The `ITouchBalls` class referenced in the call to `UIApplicationMain` is an `UIApplication` class that is responsible to create and manage the application's window.

```
<<ITouchBalls.h>>=
//
// iTouchBalls - A simple game with balls.
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
#import <UIKit/UIKit.h>

@interface ITouchBalls: UIApplication {
    UIWindow *window;
}
@end
```

```
<<ITouchBalls.m>>=
//
// iTouchBalls - A simple game with balls.
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
#import "ITouchBalls.h"
#import "BallsView.h"

@implementation ITouchBalls

<<create and set view>>

- (void)dealloc {
    [window dealloc];
    [super dealloc];
}
@end
```

The `ITouchBalls` class is not only responsible to allocate the window but also its view, which is the class that actually manages the whole game's logic. This must be done inside the `applicationDidFinishLaunching:` function called by `UIApplicationMain` when the application has done with the initialization process. This could be considered the "real" application's main.

```
<<create and set view>>=
- (void)applicationDidFinishLaunching:(UIApplication *)application {
```

The first thing I want to do in this function is to retrieve the device's screen area because I am going to need the area later to create the window and its view.

The screen area can be retrieved either using the function `bounds` or `applicationFrame` of `UIScreen` applied to the `mainScreen`. The difference between the two functions is that `bounds` will give the total device's area while `applicationFrame` returns the device's area minus the space occupied by the status bar, if shown. In this case, I want to make a full screen window, thus I'm going to use `bounds`.

```
<<create and set view>>=
    CGRect frame = [[UIScreen mainScreen] bounds];
```

Now I can make the window and its view, assign the view to the window, mark this window as the main window, and show it.

```
<<create and set view>>=
    <<create view>>
    window = [[UIWindow alloc] initWithFrame:frame];
    <<set view in window>>
```

```
[window makeKeyAndVisible];  
[view release];  
}
```

B Prefix Header

A *prefix header* is a header that the compiler automatically includes at the top of each source file. This header file is pre-compiled in order to speed up the building time. Thus, in this header I import all the other headers used by an iOS application, but only if the source being compiled is actually an Objective-C source file.

```
<<iTouchBalls_Prefix.pch>>=  
//  
// iTouchBalls - A simple game with balls.  
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>  
//  
<<license>>  
//  
// Prefix header for all source files.  
//  
#ifdef __OBJC_  
    #import <Foundation/Foundation.h>  
    #import <UIKit/UIKit.h>  
#endif
```

C Plist File

The plist file is an XML file placed inside the application's bundle that has the essential information about the program.

```
<<iTouchBalls-Info.plist>>=  
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/ ←  
    PropertyList-1.0.dtd">  
<plist version="1.0">  
    <dict>  
        <<Bundle Settings>>  
    </dict>  
</plist>
```

This file is used by iOS to retrieve information about the application. In this case it know that that application was written in an English locale.

```
<<Bundle Settings>>=  
<key>CFBundleDevelopmentRegion</key>  
<string>English</string>
```

iOS needs to know which kind of binary this bundle is holding: an application, a framework, a plug-in, etc. This game is an application, thus I set its type accordingly.

```
<<Bundle Settings>>=  
<key>CFBundlePackageType</key>  
<string>APPL</string>
```

iOS also looks in this file the name of the bundle and the string to show to the user. In both cases, I use the same string set by Xcode when building the application.

```
<<Bundle Settings>>=  
<key>CFBundleName</key>  
<string>${PRODUCT_NAME}</string>  
<key>CFBundleDisplayName</key>  
<string>${PRODUCT_NAME}</string>
```

Along with the bundle's name, iOS also shows an icon. In this case, though, I'll use an empty icon's filename because I don't have any icon for this game.

```
<<Bundle Settings>>=  
<key>CFBundleIconFile</key>  
<string></string>
```

The bundle needs to have an identifier that can be used by iOS to associate file types to the application, or store the application's settings in the appropriate place, among other things. This identifier is a reverse DNS style string where, generally, the first part is a developer's prefix — usually the developer's domain — followed by the application's name. The application name must be [RFC-1034](#) compliant.

```
<<Bundle Settings>>=  
<key>CFBundleIdentifier</key>  
<string>com.geishastudios.${PRODUCT_NAME:rfc1034identifier}</string>
```

In order to start the application, iOS must know the name of the executable to run. The executable can also be written by Xcode when building.

```
<<Bundle Settings>>=  
<key>CFBundleExecutable</key>  
<string>${EXECUTABLE_NAME}</string>
```

As a bundle can host an application for iOS or for regular Mac OS X, I need to specify that this application requires iOS (formerly called iPhoneOS) to be run.

```
<<Bundle Settings>>=  
<key>LSRequiresiPhoneOS</key>  
<true/>
```

Finally, the bundle's plist needs some extra information that I don't have any idea whatsoever what they mean. I just keep the values I've found to every other plist file.

```
<<Bundle Settings>>=  
<key>CFBundleInfoDictionaryVersion</key>  
<string>6.0</string>  
<key>CFBundleSignature</key>  
<string>????</string>  
<key>CFBundleVersion</key>  
<string>1.0</string>
```

D 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
```
