Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Črešnik

Razvoj neodvisnih video iger

Run&Roll

diplomsko delo

na univerzitetnem študiju računalništva in informatike, smer informatika

Ljubljana 2013

University of Ljubljana
Faculty of Computer and Information Science

Rok Črešnik

Indie Game Development

Run&Roll

thesis

faculty of computer and information science,
department of informatics

Mentor: doc. dr. Rok Rupnik

Ljubljana 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Črešnik

Razvoj neodvisnih video iger

Run&Roll

diplomsko delo

na univerzitetnem študiju računalništva in informatike, smer informatika

Mentor: doc. dr. Rok Rupnik

Ljubljana 2013

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Rok Cresnik, z vpisno številko **63050021**, sem avtor diplomskega dela z naslovom:

Indi Game Development

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Roka Rupnika.
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15. oktober 2013 Podpis avtorja:

Thank you!

Contents

Povzetek **1**

Abstract **5**

Prologue: The Hero is Born: iOS & Objective-C **7**

0.1 iOS 7

0.2 Objective C 9

1 Tutorial: Xcode & Frameworks **11**

1.1 Xcode 11

1.2 Cocos2d-iphone 11

1.3 Box2d 13

2 Where Does This Path Go: Movement **17**

2.1 About Controls 17

2.2 Run&Roll controls 18

3 Ow No, There Is Something In My Path: Basic Obstacles **21**

4 It's Alive, It's Alive: Animations **25**

4.1 Animations in Run&Roll 26

5 I Think I Hear Something: Sounds & Music **31**

5.1 CocosDenshion 31

5.2 Sounds in Run&Roll 33

6 The Aftermath Bragging: GameCenter **37**
6.1 Game Center 37
6.2 Implementation 38
6.3 Run&Roll: Leaderboards & Achievements 39

7 The Spoils of Battle: Monetization **43**
7.1 Mobile Gaming Revenue Models 43
7.2 Monetization in Run&Roll 45

8 To Arms My Friends: Socialization **49**

9 The Boss Fight: Competition **53**
9.1 Tracking user behavior 53
9.2 Forums, Webpages, Communities 55

Epilogue: The Story Continues: After App Submission **57**

List of Figures **59**

Bibliography **61**

List of Acronyms and Symbols

API - Application Programming Interface

GDC - Game Developers Conference

Povzetek

Neodvisne video igre imajo naziv Indie igre, ki jih razvijajo posamezniki ali manjše skupine. Diplomska naloga analizira nastanek igre Run&Roll za mobilne napreve z operacijskim sistemom iOS. Celotno delo je razdeljeno na tematske sklope, ki predstavljajo ključne pristope potrebne pri procesu izdelave mobilne igre. Delo se prične z predstavitvijo operacijskega sistema iOS, ki poganja Applove mobilne naprave, ter kratko zgodovino njegovega razvoja. Predstavljen je programski jezik Objective-C (v katerem je bila igra Run&Roll razvita), njegove glavne karakteristike in podobnosti z drugimi programskimi jeziki.
Naslednje poglavje bralca seznani z razvijalskim okoljem Xcode in pri razvoju uporabljenimi ogrodji:

- Cocos2d: ogrodje za izdelavo 2D iger
- Box2D: ogrodje za simulacijo teles v 2D prostoru ter detekcijo in razreševanje kolizij med njimi

V nadaljevanju so prikazani različni pristopi upravljanja v mobilnih igrah ter bolj podrobno analiziran način upravljanja z akselerometrom, ki je bil uporabljen pri Run&Rollu. V tem poglavju se bolj podrobno prikaže implementacija tega načina upravljanja, pridobivanje informacij iz kontrol akselerometra ter uporaba pridobljenih informacij za upravljanje akcij znotraj igre.
Poglavje "Ow No, There Is Something In My Path: Basic Obstacles" se podrobneje dotakne fizike v igri Run&Roll, kreacije fizikalnih teles, nastavljanja njihovih fizikalnih značilnosti ter akcij, ki ze zgodijo ob kolizijah.

Razkrije nam tipe fizikalnih teles uporabljenih v Run&Rollu ter posledic, ki jih sprožijo ob kontaktu z igranim junakom.
Nadaljno se prikaže uporaba sekvenc slik za animiranje glavnega junaka. Analizirani so trije tipi gibanja junaka ter logika za menjavanje med njimi. Predstavi se problem zapolnitve pomnilnika in rešitev, ki je bila uporabljena pri Run&Rollu ter sprejeti kompromis, ki je zmanjšal število potrebnih sličic brez opaznega padca kvalitete animacije.
V zaključku prvega sklopa diplomskega dela, se analizira še uporaba zvočnih efektov ter glasbe v igrah. Predstavi se ogrodje CocosDenshion (del paketa cocos2d), ki poenostavi predvajanje zvočnih učinkov ter glasbe. Poglavje podrobno analizira zvočne učinke v Run&Rollu, ki so razdeljeni v tri sklope, poudari pa tudi pomembnost prednaložitve zvočnih datotek pred začetkom igre, kar izboljša igralno izkušnjo.

Druga polovica diplomskega dela se osredotoči na podporne elemente igre. Prvo poglavje tega sklopa predstavi način točkovanja igralne seance, implementacijo lestvic in dosežkov s pomočjo Applove storitve GameCenter ter pošiljanjem rezultatov iz mobilne naprave na GameCentrove strežnike. Krajši odsek je namenjen tudi predstavitvi storitve GameCenter in vplivu, ki ga ta storitev ima na igralce mobilnih iger. Cilj razvijalcev mobilnih iger je pokriti stroške nastale z razvojem. Tega se ekipe lotevajo na različne načine, ki so predstavljeni v poglavju “The Spoils of Battle: Monetization”. Podrobneje je predstavljen tip brezplačnih iger z trgovino, v katerega spada tudi analizirana igra. Opisan je postopek kreacije prodajnih artiklov na portalu iTunesConnect ter potrebni koraki za verifikacijo in potrjevanje nakupov za denar. Predstavljena je trgovina Run&Rolla, njena segmentacija in načina delovanja. Del trgovine je tudi sekcija, kjer si igralci lahko sledijo igri na platformah Facebook, YouTube in Twitter. V zameno pridobijo vnaprej določen znesek kovancev, ki jih lahko porabijo za nadgradjo igralnih elementov. Poleg tega je opisan še en način pridobivanja igralne valute - z prikazovanjem doseženega rezultata na socialnih omrežnjih. Igralec preko različnih mrež

prikaže svoj dosežek in s tem izzove prijatelje, da ga premagajo. Nadalje je izpostavimo pojem uporabniških metrik ter predstavitev ogrodja Flurry, ki omogoča pridobivanje le-teh. Opisan je postopek implementacije ter zajemanja različnih dogodkov. Ti dogodki se preko spletnega vmesnika lahko nadaljno povezujejo v lijake, uporabniške poti. Z analizo pridobljenih podatkov lahko pripomoremo k večjemu uspehu produkta. Pomemben aspekt uspeha igre je tudi podpora neodvisnih razvijalskih skupnosti. V delu je predstavljena struktura in pomembnost teh skupin ljudi, ki lahko z nasveti in kritikami pomagajo h boljšemu produktu in večjemu številu igralcev.

Ker je postopek izdelave igre zelo dolg in kompleksen proces, se v diplomskem delu nisem spuščal v pretirane podrobnosti, vendar sem se poizkusil dotakniti vseh pomembnih aspektov in problemov s katerimi smo se pri razvoju srečali.

Ključne besede:

neodvisen razvoj iger, mobilne igre, iOS, Objective-C, cocos2d, Box2D, mobilne naprave, monetizacija, viralnost, animacije, zvok, detekcija kolizij, razrešitev kolizij, socialna omrežja

Abstract

Indie games are independent video games created by individuals or small teams. This thesis analyses the process behind the creation of the iOS game Run&Roll. Through the following chapters, the readers will familiarize themselves with various tools and concepts used in the realization of the final product. The thesis begins with an outline of the basics of the iOS operating system and Objective-C language, which was used in the development of Run&Roll. Various frameworks and their use in the game are explained and displayed. Various concepts of game creation and their implementation are introduced. The first chapters focus on concepts of movement in mobile gaming, interaction with different objects (physics) and the basics of animation and sound. Each chapter is equipped with short code snippets, depicting the realization of the concepts in question. In the second part of the thesis, focus is given to concepts of monetization (inApp purchases, advertisements) and virality (social network integration and multiplayer). In the conclusion, concepts of competition and marketing in the mobile gaming field are discussed. Since the development of a game is a very long and complex process, the subject is not explained in high detail. Instead, the most important aspects and problems we encountered in the production of the game are explained.

Key words:

indi game development, mobile games, iOS, Objective-C, cocos2d, Box2D, mobile devices, monetization, virality, animation, sound, collision detection,

collision resolution, social networks

Prologue: The Hero is Born: iOS & Objective-C

2012 was an interesting year, and not because we lost a number of industry giants, but because the number of smartphones has exceeded the combined number of PCs and notebook PCs. This has created a trend - a lot of developers changing their focus to the booming sector of mobile devices. A mobile device is a small, hand-held computing device, typically equipped with a display screen with touch input and/or a miniature keyboard, and weighing less than 0.91 kg. In our case, the term "mobile device" describes either a smart phone or a tablet PC. Figure 1 depicts the market shares of various companies in the smart phone of mobile devices. As depicted, Android devices hold the largest market share, followed by Apple's iOS. This thesis focuses on Apple's segment of the mobile device market. This segment includes iPhone, iPad and iPod Touch devices.

0.1 iOS

Apple's i-devices are all running the iOS (previously iPhone) operating system, which was introduced by Apple in 2007 on the first iPhone and iPod Touch devices. Since the launch date in 2007, iOS has gone through major changes, some of the more notable ones being:

- App Store: introuduced with iOS 2.x on 11th of July, 2008.

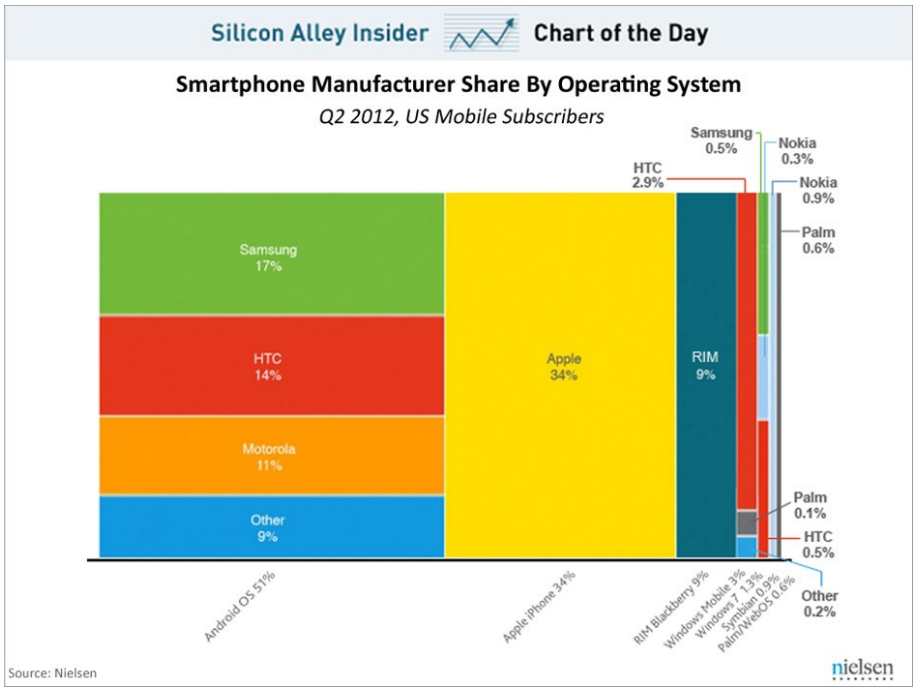


Figure 1: Smartphones Manufacgturers Share by Operating System (Source:Silicon Valley Insider)

- Copy-Paste functionality, MMS: introudced with iOS 3.x on 17th of June, 2009.

- Apple's operating system renamed to iOS, dropped support for some devices, added iPad compatibility: introudced with iOS 4.x on 21th of June, 2009.

- iMessage, Retina display, Siri: introudced with iOS 5.x on 12th of October, 2011.

- Larger screen, new maps, Facebook integration (introduced with iOS 6.x on September 19th, 2012).

0.2 Objective C

Objective-C is the primary language, used to develop software for iOS and OSX. It inherits syntax, primitive types and flow control statements from the C programming language, while adding object-oriented capabilities. In Objective-C, the most work is done with objects, instances of Objective-C classes, which can be provided by Cocoa/Cocoa Touch or can be written on one's own. The main designing concepts, commonly used in iOS development are:

- Categories: Instead of creating a new class to add additional capabilities, Objective-C allows the defining of categories which add custom behaviour to any pre-existing class (even to classes for which one does not have the original source code).
- Protocols: Objective-C uses protocols to define a set of required or optional methods that are not tied to a specific class, but are implemented on its delegate. Any class can adopt a protocol, but must first provide implementation for all of the required methods of the protocol.
- Blocks: A block represents a unit of work. It encapsulates a block of code with a captured state, which makes it similar to closures in other programming languages. Blocks are often used to simplify common tasks, such as collection enumeration, sorting, and testing. They also make it easier to schedule tasks for concurrent or asynchronous execution.

Level 1

Tutorial: Xcode & Frameworks

1.1 Xcode

Xcode is the Integrated Development Environment (IDE) containing developmental tools, developed by Apple. It was released in 2003 and is mainly used for developing software for iOS and OSX. Xcode is free to use and can be obtained from the Mac App Store. Figure 1.1 is a snapshot of the Xcode application. The left section or window is called the Navigator and lists a project's file structure. The bottom section is the Debug area, further divided into two areas - the Variable view and the Console. To the right are the Utilities. The top section is used for building or analysing an application and different interface views.

Xcode comes with a simulator for trying out builds (for developers that do not own their own devices or do not have Apple's developer license). Have I mentioned one cannot deploy builds on a real device without owning a valid developer license?

1.2 Cocos2d-iphone

Cocos2d is an open source framework for building 2D games. The original framework was written in Python, but has since been ported to other

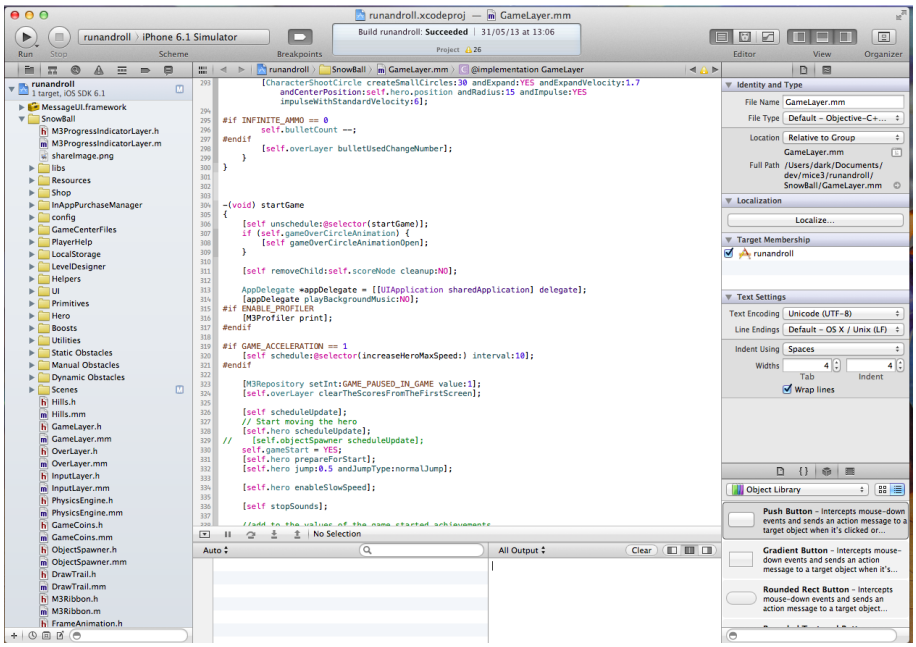


Figure 1.1: Apples IDE - Xcode

platforms, one of them being Objective-C. The Application Programming Interface (API) is integrated with Chipmunk and Box2d physics engines (more on Box2d in the following section). Cocos2d is widely used by developers all over the world; many prefer it over Objective-C. Here are some of the reasons for that:

- It is free: Without paying, one is allowed to create free applications for iPod, iPad, iPhone and even the Mac OS X platforms.
- It is open source: The whole framework is open to the community, allowing developers to read, edit and change whatever they want, making the Cocos2d framework both extensible and flexible.
- It is Objective: The framework was rewritten in Objective-C, Apple's native programming language.
- It has Physics: As mentioned previously, there are two physics engines integrated into Cocos2d - Chipmunk and Box2d. The main difference

between them is the language they are written in. Box2d is written in C++, while Chipmunk is written in C. Most developers choose Box2d, because it is better documented and is object oriented.

- It is less complex: One of the major benefits of Cocos2d is the way it hides complicated OpenGL code. Most of the graphics are drawn using simple sprite classes, created from image files. To put it frankly: A sprite is a texture that can have scaling, rotation, and colour applied by changing the corresponding Objective-C properties of a CCSprite class. At the same time, Cocos2d enables advanced developers to write their own OpenGL code, making the framework appreciated by both new and advanced users.

- It has a great community: The Cocos2d community is large and extremely active. The developers are quick to answer questions and willing to share their knowledge and information (one of the main sources of information is the Cocos2d forum: http://www.cocos2d iphone.org/forum).

After a game has been finished and released on the App Store, it is even possible to promote it on the Cocos2d web page for free! Besides being the mostly commonly used game developing platform for iOS, Cocos2d ports exist on other developing platforms as well. For example: Android, Windows, JavaScript. Even though all of the mentioned ports share the same name and design philosophy, they are actually written in different languages by different authors, and are very different from the iOS version of Cocos2d. This means porting to other platforms also demands more work. Table 1.1 lists the Cocos2d game engines, frequently updated and stable enough for production use.

1.3 Box2d

Let us take a closer look at the Box2d physics engine. The engine is written in C++ and was developed by Erin Catto. It was presented for the first time

Name	Language	Platform	Web Site
cocos2d-iphone	Objective-C	iOS, Mac OS X	`www.cocos2d-iphone.org/`
cocos2d-x	C++	iOS, Android, Windows	`www.cocos2d-x.org/`
cocos2d-javascript	JavaScript	Web browsers	`www.cocos2d-javascript.org/`
cocos2d-android-1	Java	Android	`http://code.google.com/p/cocos2d-android-1/`
cocos2d	Python	Mac OS, Windows, Linux	`http://cocos2d.org`

Table 1.1: Most popular Cocos2d engine ports

at the Game Developers Conference (GDC) in 2006 (at that time, it was named Box2D Lite) and has been in active development ever since. Box2D is distributed with Cocos2d in light of its recent popularity. Box2D performs constrained rigid body simulation. It can simulate bodies composed of convex polygons, circles, and edge shapes. Bodies are joined together with joints and affected by various forces. The engine also applies gravity, friction, and restitution. Box2D's collision detection and resolution system is conducted in three phases: Incremental sweep and prune broad-phase, continuous collision detection unit, and stable linear-time contact solver. These algorithms allow efficient simulations of fast bodies and large stacks, without missing collisions or causing instabilities. Box2D consists of:

- Bodies: Bodies are the fundamental objects in Box2D. They have properties that define their behaviour (mass, velocity, rotational and angular velocity, inertia, location, and angle). However, even with the previously defined properties, we still do not know how the object looks like or how it will react upon a collision. For that, we need fixtures.

- Fixtures: Fixture defines the size, shape and the material properties

of an object. One body can have multiple fixtures that will affect the body's centre of mass. When two (or more) bodies collide, their fixtures are used to determine the outcome of the collision.

- Worlds: A world is the main entity in which all the Box2D bodies reside. When creating or destroying a body, a function of the world object is called to perform the required task. This means that the world entity manages all the allocations of the objects within. The world entity is important and is used for defining gravity, tuning the physics simulation, finding fixtures in a given region, and finding intersected fixtures.

- Joints: Box2D has a number of different joint types (revolute, distance, prismatic, line, weld, pulley, gear, mouse), used to connect bodies together. These joints are used to simulate the interaction between objects, to form hinges, ropes, pistons, chains, etc. Although each joint has different functionalities, they have some common features. Every joint connects two bodies, and has a setting that determines if those bodies are able to collide with each other.

- Collisions: The core of Box2D are collisions. As mentioned earlier, when two bodies collide, we use their fixtures to determine the outcome. Collisions can occur in different situations and have a lot of information that can be used in the game logic. Important collision information includes: The beginning and ending of collision, points in which the fixtures are colliding, the normal vector between the colliding fixtures, and energies involved in collision and resolution.

The frameworks described above were used in creating Run&Roll, and will be analysed in the following chapters.

Level 2

Where Does This Path Go: Movement

2.1 About Controls

The way our hero moves depends primarily on the type of the game. The most commonly used movement types on mobile devices are:

- Accelerometer control: This control type uses the amount of X, Y, Z axis acceleration to perform an action. We can define which axis will be used for a certain action.

- Virtual gamepad: This is a virtual representation of a gamepad on the screen of our device. Virtual gamepads come in many shapes and sizes, customized to meet our needs.

- Touch/drag control: Movement is achieved by touching an object and dragging it to a new position (this type is used mainly in board games like Chess, Backgammon, etc.).

These are the main control types, using which the player can interact with a game. Besides these, there are also many others, more exotic ways of control.

2.2 Run&Roll controls

Run&Roll is an endless runner game in which we control Joe the Hedgehog (henceforth referred to as the "Hero"), avoiding obstacles and collecting as many coins and boosts as possible. The games becomes faster the longer it lasts, forcing shorter reaction times and thus making it harder for the player to cope with the game. Because the Y axis speed is controlled by the game system, a player's only task is to move the Hero left and right, which is achieved with the help of the accelerometer. We decided on an accelerometer controlled game due to factors as:

- Ease of use: Movement is achieved by tilting the phone left and right.
- Screen space: No additional movement buttons are needed.
- Advanced use of available hardware.
- Easy implementation.

Enabling the accelerometer is a straight forward process. All we need to do is to initialize it and then bind the accelerometer values to our Hero. The initialization and movement of our Hero are shown in the code snippet below.

```
- (id)initAcc
{
 if ((self = [super init])) {
        self.accelerometer = [UIAccelerometer sharedAccelerometer];
        self.accelerometer.updateInterval = .1;
        self.accelerometer.delegate = self;
 }

 return self;
}

- (void)accelerometer:(UIAccelerometer *)accelerometer
     didAccelerate:(UIAcceleration *)acceleration
```

```
{
        float absAccelerationX = fabs(acceleration.x);
        float xDirection = 1;
        if (acceleration.x < 0) {
            xDirection = -1;
        }
        float calibratedAccX = MIN(absAccelerationX*1.9, 1) * xDirection;

        float angle = M_PI+M_PI_2 + M_PI_2 * calibratedAccX;
        hero.accelerometerAngle = angle;
}
```

The *initAcc* function initializes the accelerometer, sets its update interval to 0.1 seconds and declares the accelerometer's delegate. The function *accelerometer:didAccelerate* detects changes in the accelerometer. The accelerometer measures acceleration in all three directions, but because our hero can only move left or right, we use the data for the X axis only. The code in the snippet above takes into account the adjustments we have made to enhance the movement of our Hero, making him more agile.

Level 3

Ow No, There Is Something In My Path: Basic Obstacles

In this chapter we will discuss obstacles, their creation and how they respond to collisions. Run&Roll features three main obstacle types. These are:

- Static obstacles: These are obstacles that do not move. In Run&Roll we further divide them into:
  - Obstacles the Hero can collide with (rocks, trees, crystals, hills, etc.).
  - Obstacles the Hero can fall into (rivers, holes, etc.).
  - Obstacles the Hero must jump over.
- Dynamic (moving) obstacles (monsters).
- Collectable objects: This group includes items our Hero can collect:
  - Coins: Coins are the game's currency. The more we collect, the more we can spend in the Run&Roll shop, where we can improve our Hero's performance.
  - Boosts: These are special bonus items, spread around the game's environment that enhance our Hero's capabilities, whether making

him fly, destroy objects or collect coins faster. Every boost is triggered instantly and lasts for a fixed amount of time. The duration of a boost can be increased by purchasing boost upgrades in the Run&Roll shop. The Run&Roll game features five different boosts (health, speed, magnet, meteor, and tank boost).

– Utilities: Unlike boosts, utilities can be used on demand, but only when their trigger conditions are met. Utilities can either be bought or collected. There are three different utilities in Run&Roll: Resurrection, head-start, and extra health.

Now that we know more about obstacles in Run&Roll, let us talk code! As explained before, obstacles behave differently. Different types of obstacles have differing properties, set to make them behave in a way we desire. In Box2D, these properties are defined in the body's fixture and are called mask and category bits. They define the way a body reacts upon collision, either stopping the colliding objects, or letting it pass uninterruptedly. To see how a Box2D body is created, look at the code snippet below. It contains the code needed to create a simple static obstacle – a rock.

```
– (void)createBox2dBodyWithUserData: (id) userData
{
        CGPoint startPosition = self.position;

        b2BodyDef bd;
        bd.type = b2_staticBody;
        bd.linearDamping = 0.f; // between 0 and 0.1
        bd.fixedRotation = true;
        bd.position.Set(startPosition.x/PTM_RATIO, startPosition.y/PTM_RATIO);
        bd.allowSleep = false;
        bd.userData = userData;
        _body = _world->CreateBody(&bd);

        b2CircleShape shape;
        shape.m_radius = 10;
```

```
        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 5;
        fd.friction = 0; // friction
        fd.restitution = 0.0; // bounce effect
        fd.filter.categoryBits = self.box2dCategoryBits;
        fd.filter.maskBits = self.box2dMaskBits;
        fd.isSensor = false;
        _body->CreateFixture(&fd);
}
```

A reasonable amount of code is needed to create a simple obstacle. There is a lot of discernable information, hidden in the code above. The first chunk defines the body, its position, its rotational abilities, etc. After the body has been created, we need to create its fixture. As stated before, a fixture determines how the body reacts upon collision. First, we define the body's shape and size; our rock is of circular shape with a radius of 10 units. Then we set the density, friction and restitution, which determine the colliding behaviour.

- Density is used to compute properties of a body's mass. It can either be zero (0) or positive (1).
- Friction is used to make objects slide along each other. It is usually set between zero (0) and one (1). The value of 0 turns the friction off, and the value of 1 makes the friction strong.
- Restitution is used to make objects bounce. Restitution is usually set between zero (0) and one (1). Given the value of 0, the object will not bounce, and given the value of 1, the object's velocity will be inverted without losses (perfect elastic collision).

Lastly, we set the mask and category bits and the *isSensor* property. This is called collision filtering and allows us to prevent collisions between fixtures.

Box2D supports 16 collision categories. For each fixture we can specify which category it will belong to. We can also specify other categories a fixture can collide with. For example, we can set up a multiplayer game in which players cannot collide with each other, monsters cannot collide among themselves, but players and monsters can. The *isSensor* property defines whether a body's collision is calculated. If *isSensor* is set to true, a body will not collide with any other objects (it behaves as if its mask bits are set to zero).

Level 4

It's Alive, It's Alive: Animations

Making objects appear alive can be achieved with the use of animations. An animation is a rapid display of a sequence of images that creates the illusion of movement. Cocos2D comes with classes that enable developers to create simple animations without having to do too much work. The simplest method of making animations in Cocos2d is importing a sequence of images into an Xcode project, applying them to an object and then exchanging them on the desired object. However, there is a better way of creating animations in Cocos2d - by using sprite sheets. A sprite sheet is a gigantic image, containing sprite images. Sprite sheets always come in pairs. A pair is composed the following files:

- Sprite sheet file: This is a large image, containing all animation frames.

- Plist file: The plist file contains information on individual sprite boundaries. With the information provided by the plist file, we can retrieve individual images from the sprite sheet file and create animations.

4.1 Animations in Run&Roll

In Run&Roll animations were used to make our Hero run, jump and roll. For that purpose, we created a sprite sheet file Hero.png, shown in figure 4.1, with its pair Hero.plist. Because our Hero's movement types differ and he can move in different directions, a systematic naming convention that can utilize the different types and angles of movement was needed. Run&Roll Hero animation naming convention:

`Hero_[movementType]_[directionalAngle]_[frameCount].png`

Where:

movementType is either "walk" or "roll".

directionalAngle describes the angle of our Hero's movement. Due to memory limitations, a limited number of angles had to be selected. For that reason, we only use certain angles (135, 180, 225, 240, 255, 270, 285, 300, 315, 360, and 45 degrees). If you look at the angle selection closely, you will notice that the angular difference is not always the same. This is the result of our Hero normally moving somewhere between the angles of 255 and 315 degrees, where we made the animation look more detailed. If the Hero's movement angle is lower than 225 or higher than 315 degrees, his movement can be utilized, which does not happen that often, we use less frame sequences to animate the Hero's movement.

frameCount indicates the frame number of the movement animation. Each movement angle has 16 frames for each movement type.

This brings us to the total frame count of (2 *11*16) 352 images that create the illusion of movement. To make the animation appear smoother, we used sprite rotation to cover up blind spots, these being the animation angles not included in the sprite sheet file. For example, if the Hero is moving at an angle of 265 degrees, the animation for the 270 degrees angle is applied, and the sprite is rotated for additional 5 degrees, creating the illusion of the Hero

moving in the desired direction. The code snippet below shows how our Hero's animation was created, and the logic for angle and movement type changes.

```
// Part 1: FrameAnimation.m
- (id)initWithAnimationName: (NSString *) animationName animationType:
(NSString *) type angle:(int)angle withNumberOfFrames: (int) frameCount
{
    if ((self = [super init])) {
        self.isRepeating = YES;
        self.animationName = animationName;

        CCSpriteFrameCache *spriteCache = [CCSpriteFrameCache
    sharedSpriteFrameCache];
        NSString *plistFileName = [NSString stringWithFormat:@"%@.plist",
    animationName];
        [spriteCache addSpriteFramesWithFile:plistFileName];
        NSString *imageName = [NSString stringWithFormat:@"%@.png", self.
    animationName];
        CCSpriteBatchNode *spriteSheet = [CCSpriteBatchNode batchNodeWithFile:
    imageName];

        self.animationType = type;
        self.angle = angle;

        NSString *currentFrameName = [NSString stringWithFormat: @"%@%@%i_0.
    png", self.animationName, self.animationType, self.angle];

        self.frame = [CCSprite spriteWithSpriteFrameName: currentFrameName] ;

        [self addChild:self.frame];
        self.currentFrame = 0;
        self.frameCount = frameCount;

        self.isPlaying = NO;
    }
    return self;
}
```

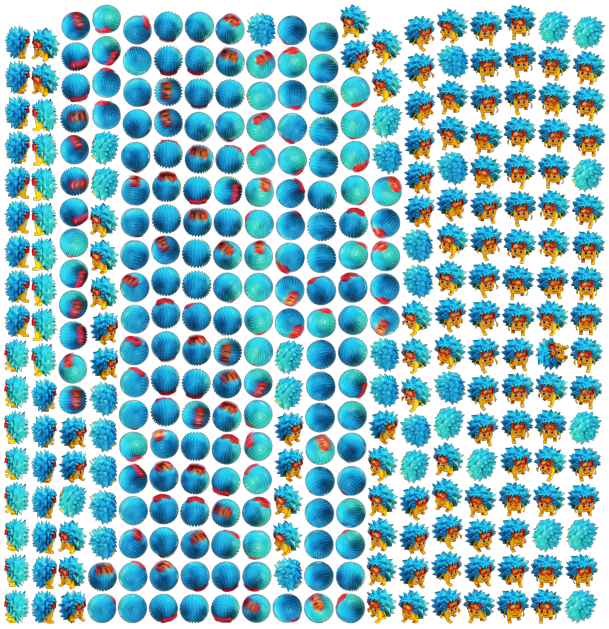
```
// Part 2: Heros implementation file we create the animation:
self.moveAnimation = [[FrameAnimation alloc] initWithAnimationName:@"Hero"
    animationType: @"walk" angle:270 withNumberOfFrames:16];

// Part 3: we need to update the displayed image each frame.
// The bellow function does that
- (void)updateDisplayedFrame {
    float hero3dRotation = self.prevAccelerometerAngle;
    self.prev3dRotation += (hero3dRotation-self.prev3dRotation)/10;

    float speedAngle = CC_RADIANS_TO_DEGREES(newAngle);
    FrameAxis result = [self getFrameAxisFromAngle:speedAngle detailedAxis:YES];

     totalDistance += ccpDistance(self.position, self.prevPosition));
    [_currentAnimation setAngle:result.axisAngle];
    int frameNum = totalDistance/8)% _currentAnimation.countFrames;
    [_currentAnimation gotoAndStop:frameNum];

    float angleCorrection = (speedAngle - result.axisAngle);
    if (angleCorrection > result.coeficient) {
        angleCorrection = result.coeficient;
    } else if (angleCorrection < -result.coeficient) {
        angleCorrection = -result.coeficient;
    }
    _currentAnimation.rotation = -angleCorrection;
}

// Part4: animation type changes
- (void)changeAnimation:(AnimationType) type
{
    self.animationType = type;
    [_currentAnimation pause];

    switch (self.animationType) {
        case walkingAnimation:
            [_currentAnimation changeAnimationTypeTo:@"_walk" withCountFrames
    :16];
```

```
            break;
        case runningAnimation:
            break;
        case tankAnimation: {
            if (self.state == tankState) {
                [self.graphicsContainer removeChild:self.moveAnimation cleanup:NO];
                [self.graphicsContainer addChild:self.tankAnimation];
                _currentAnimation = self.tankAnimation;
            } else {
                [self.graphicsContainer removeChild:self.tankAnimation cleanup:NO];
                [self.graphicsContainer addChild:self.moveAnimation];
                _currentAnimation = self.moveAnimation;
            }
            break;
        }
        case airRollingAnimation:
            [_currentAnimation changeAnimationTypeTo:@"_roll" withCountFrames
    :15];
            //[_currentAnimation play];
            break;
        case rollingAnimation:
            [_currentAnimation changeAnimationTypeTo:@"_roll" withCountFrames
    :15];
            break;
        default:
            break;
    }
}
```

Explanation of the above code:

- Part 1 is from our FrameAnimation class. It creates a CCSpriteFrame and handles the frame changing logic.
- Part 2 shows how FrameAnimation is created in our Hero class.
- Part 3 handles the angle changing and occurs every update (each frame)

- Part 4 handles animation type changes and occurs each time our Hero jumps or collects a boost that influences his movement type (Speed-Boost, Tank-Boost, Meteor-Boost).



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST
1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>frames</key>
    <dict>
      <key>Hero_roll_135_0.png</key>
      <dict>
        <key>frame</key>
        <string>{{270,1072},{51,50}}</string>
        <key>offset</key>
        <string>{0,10}</string>
        <key>rotated</key>
        <false/>
        <key>sourceColorRect</key>
        <string>{{67,104},{51,50}}</string>
        <key>sourceSize</key>
        <string>{185,278}</string>
      </dict>
      <key>Hero_roll_135_1.png</key>
      <dict>
        <key>frame</key>
        <string>{{328,913},{51,52}}</string>
        <key>offset</key>
        <string>{0,11}</string>
        <key>rotated</key>
        <true/>
        <key>sourceColorRect</key>
        <string>{{67,102},{51,52}}</string>
        <key>sourceSize</key>
        <string>{185,278}</string>
      </dict>
      <key>Hero_roll_135_10.png</key>
      <dict>
        <key>frame</key>
        <string>{{385,911},{51,52}}</string>
        <key>offset</key>
        <string>{1,14}</string>
        <key>rotated</key>
        <true/>
        <key>sourceColorRect</key>
        <string>{{68,99},{51,52}}</string>
        <key>sourceSize</key>
        <string>{185,278}</string>
      </dict>
      <key>Hero_roll_135_11.png</key>
      <dict>
```

Figure 4.1: Hero - Sprite Sheet and a part of the coresponding plist file

Level 5

I Think I Hear Something: Sounds & Music

So far, we have covered:

- Movement: We can move our Hero with the use of accelerometer sensors.
- Physics: Our Hero can collect objects or collide with them.
- Animation: Our Hero appears 3D, depending on his movement type, speed and direction.

And what else do we need in a game? Background music and sound effects, of course! In the following chapter, focus is given on these.

5.1 CocosDenshion

Audio support is not an integral part of Cocos2D, it instead falls under the domain of CocosDenshion, which is a third-party addition for Cocos2D and is fortunately distributed with it. Because this is a Cocos2d addition, we need to import the right header files wherever we want to use audio playback functionalities. This is achieved with the use of the following line of code:

```
#import "SimpleAudioEngine.h"
```

CocosDenshion is a wrapper class that hides the complex low level coding and provides us with all the basic sound playing functionalities. These are:

- Play effect:

  ```
  [[SimpleAudioEngine sharedEngine] playEffect:@"effect.mp3"];
  ```

- Play background music:

  ```
  [[SimpleAudioEngine sharedEngine] playBackgroundMusic:@"music.mp3"];
  ```

- Pause background music:

  ```
  [[SimpleAudioEngine sharedEngine] pauseBackgroundMusic];
  ```

- Play background music from the start again:

  ```
  [[SimpleAudioEngine sharedEngine] playBackgroundMusic:@"music.mp3"];
  ```

- Preload background music/effect:

  ```
  [[SimpleAudioEngine sharedEngine] preloadBackgroundMusic:@"music.mp3"];
  [[SimpleAudioEngine sharedEngine] preloadEffect:@"effect.mp3"];
  ```

CocosDenshion does not only provide us with means of simple audio playback, it also provides us with means of controlling sound pitch, pan and gain levels.

- Pitch [0.5 to 2.0]: Pitch is the "note" of the sound. A higher pitch value produces sound at a higher note. A lower value makes the sound "deeper". 1.0 is the pitch of the original file.
- Pan [-1.0 to 1.0]: Pan is the stereo effect. Levels below zero raise the loudness of the left speaker and reduce the loudness of the right, while levels above zero raise the loudness of the right, and reduce the loudness of the left speaker (if possible due to hardware restrictions).
- Gain [0.0 and up]: Gain is the volume. 1.0 is the volume of the original file.

Bearing this in mind, let us take a look at the implementation of sounds and music in Run&Roll.

5.2 Sounds in Run&Roll

The sounds in Run&Roll were composed by Peter Penko, a world renowned composer, who has worked with groups such as: Laibach, Siddharta, Terra Folk, Witt*, Coptic Rain, and others. Sounds in Run&Roll can be differentiated into three groups:

- Background music: The game features two background soundtracks, one intended for gameplay and the other for the menus (main screen, score scene, shop, etc.). Soundtracks can be easily switched, using the following code:

```
- (void)playBackgroundMusic:(BOOL)forMenu
{
    NSString *fileName;
    if (forMenu) {
        fileName = @"backgroundMusicMenu.mp3";
        if (_isMenuMusicPlaying) {
            return;
        }
```

```
        [[SimpleAudioEngine sharedEngine] stopBackgroundMusic];
        _isMenuMusicPlaying = YES;
    } else {
        fileName = @"backgroundMusic.mp3";
        [[SimpleAudioEngine sharedEngine] stopBackgroundMusic];
        _isMenuMusicPlaying = NO;
    }

    [[SimpleAudioEngine sharedEngine] playBackgroundMusic:fileName];
}
```

- Effects, played once (collision effects, different pickup sounds): These sound effects last a short period of time and are played only once. Their playback requires no altering. Because there are different effects for different situations, we have created a function with two parameters (effect base name, number of effects).

```
+ (CDSoundSource *)playEffect: (NSString *)effectName numberOfEffects:(int)
    number
{
    NSString *effectName = [self getEffectName:effectName
 andNumberOfEffects:number];
    NSString *fullName = [effectName stringByAppendingString:@".mp3"];
    return [ self playSound:fullName];
}
+ (NSString *)getEffectName:(NSString *)name andNumberOfEffects:(int)
    number
{
    if (number > 1) {
        int randomNumber = arc4random() % number;
        return [NSString stringWithFormat:@"%@%i", name, randomNumber];
    } else {
        return name;
    }
}
```

The first function receives the arguments and calls *getEffectName:andNumberOfEffects:* which generates the name of the effect out of the received arguments. The effect is then played and returned to the caller.

- Longer lasting effects (monster follower effect, wall collision effect): These are effects, played for a longer period of time and require extra logic. One of them is the monster follower effect. This sound is triggered by a monster, chasing our Hero; this happens, when the monster gets in range of our Hero. The loudness of the sound either increases, when the monster approaches the Hero, or decreases, when distance between them is increased.

```
// volume of followerSound according to the distance
self.followerSound.gain = 1 - (distance - 10) / 190;
```

In order for the above to be possible, we must save a reference to the effect currently being played in one of our object's properties. With the above example, this is achieved by using the following lines of code:

```
// here we create the property for our CDSoundSource
@property (nonatomic, retain) CDSoundSource *followerSound;
// here we asign the CDSoundSource to our previously created property
self.followerSound = [AppDelegate playEffect:@"monster" numberOfEffects:2];
```

Before moving on, there is one more subject I would like to cover. It is considered as good practice to preload all the needed sound effects, before they are played. If we fail to do so, the players might experience a slight gameplay lag the first time a certain sound effect it played. For that reason, we have created a Sound preloading class, which preloads all of our sound files before the gameplay actually begins. The following code snippet does just that:

```
NSArray *extensions = [NSArray arrayWithObjects:@"mp3", nil];
 for (NSString *extension in extensions) {
     NSArray *paths = [[NSBundle mainBundle] pathsForResourcesOfType:
 extension inDirectory:nil];
     for (NSString *filename in paths) {
         [loader addResources:filename, nil];
     }
 }
```

The extensions array can contain multiple entries for different types of files we want to preload. In our case, it was used to preload sound files.

Level 6

The Aftermath Bragging: GameCenter

In order to encourage competitiveness in Run&Roll, we have ensured the possibility of measuring a player's performance. Playing competitive games, everyone wants to know how good he or she did, not only in comparison to his or her own performances, but against other players as well. And this is only one of the functionalities Game Centre offers.

6.1 Game Center

Game Centre is an online multiplayer social gaming network released by Apple on September 8th, 2010. Prior to Game Centre's release, the market was dominated by various service providers (OpenFeint, Plus+, AGON Online, Scoor- loop, and others), which led to a inconsistent user experience. Game Centre was announced during the iOS 4 preview and has since been implemented into the majority of iOS games. Using Game Centre, users can connect with friends by sending out friend requests, organizing online game sessions, and much more. Today, many iOS games use Game Centre, but not all of them use every one of its features. Applications can include any or all of the following features, supported by Game Centre:

- Leader boards: Here, one can compare scores with his or her friends and other players from around the world. A game can have multiple leader boards, each covering a certain aspect of the game.
- Achievements: Here, the achievements and goals a player can unlock or accomplish are shown and their progress indicated. For each unlocked achievement, players are awarded points, showcasing their progress in the game.
- Multiplayer: It is possible for players to host games themselves and play with their friends, and to play against random opponents.

6.2 Implementation

After a game's mechanics is developed, the Game Centre implementation comes into play. The implementation is conducted through the following steps:

- Create an new application on iTunes Connect, using the game's bundle ID. Then enable Game Centre for the created application; enabling this setting in the iTunes Connect record authorizes the Game Centre service to connect to your game.
- Create an explicit application ID, using your game's bundle ID. Enable Game Centre with this application ID; this authorizes the application on the device to contact Game Centre's servers.
- Create a new provisioning profile, using this new, explicit application ID.
- Test to ensure you can build and sign your game using this profile.
- Add the Game Kit framework to your game.
- Import the GameKit/GameKit.h header into your class.

All the desired leader boards and achievements need to be created on the iTunes Connect webpage, before they can be used in game. Game Centre allows for a total of 100 points to be distributed between all existing achievements. The points do not need to be distributed equally among the achievements, and the total sum of points distributed must not pass 100. Each achievement requires a selected goal to be reached for the achievement to become unlocked. There are two distinguishable types of achievements:

- Single session achievements: In order to unlock these, the player must accomplish the given goals in a single playing session.
- Cumulative achievements: The progress of accomplishing these achievements is measured over multiple playing sessions.

6.3 Run&Roll: Leaderboards & Achievements

Two aspects of Game Centre were implemented into and are used in Run&Roll – leader boards and achievements. Leader boards inspire competitiveness; there, players can share their scores and compare them to others' (more on this topic in chapter 8). In Run&Roll, a single leader board, named: How far did you go? was implemented. The leader board ranks players according to their score, calculated from the run's length, number of collected coins, destruction factors, and the number of jumps. After each game session, the score is calculated and sent to Apple's Game Centre server. The following code snippet demonstrates the use of the "call" function:

```
// score - play session score
// category - Game Center's Leaderboard ID
- (void)reportScore: (int64_t) score forCategory: (NSString*) category
{
 GKScore *scoreReporter = [[[GKScore alloc] initWithCategory:category] autorelease];
 scoreReporter.value = score;
 [scoreReporter reportScoreWithCompletionHandler: ^(NSError *error) {
   [self callDelegateOnMainThread: @selector(scoreReported:) withArg: NULL error:
    error];
```

```
    }];
}
```

Achievements in Run&Roll can be divided into the following groups:

- Distance achievements: Unlocked when the player reaches a certain distance in a single playing session (3.000 meters, 6.000 meters and 10.000 meters).
- Coin achievements: Unlocked when the player collects a certain amount of coins over multiple playing sessions (5.000, 10.000, and 40.000 coins collected).
- Miscellaneous achievements:
  - "The Unlucky One": The player dies 1000 times.
  - "Speeeed": Travel 100 meters in speed mode.

Now let us look closer into the Coin achievement function call. Every time our hero collects a coin, the following takes place:

- The total sum of collected coins is increased by one.
- The sum is compared to the achievement goal.
- If the goal is reached, the user is notified of his newly unlocked achievement (as seen in figure 6.1)

We have created sliding notifications for a non-obstructive display of newly unlocked achievements, which appear on the top of the screen and disappear after a short period of time.

Figure 6.1: Achievement triggered

Level 7

The Spoils of Battle: Monetization

"There are now over 1 billion active smartphones and tablets using apps around the world every month. And of all the apps consumers use, games command more than 40% of all time spent. Looking at revenue, games also dominate. Today, for example, 22 of the top 25 grossing apps in the U.S. iTunes App Store apps are games. Gamers spend money, and game makers are in love", Dan Laugins, Flurry.
Games, as most other products, need to cover the costs of work that was invested into their creation. According to Dan, they are really good at it! As sure as Dan seems of this, the reality is that making a profitable game is far from being a simple and easy task, especially for indie game developers. How they cope with problems that arise in developing a profitable game, is the topic of the following subchapter.

7.1 Mobile Gaming Revenue Models

First, let us take a look at the most common revenue models in mobile gaming:

- **Pay-to-play**: This is the most common revenue model in the field of

mobile gaming. First, players pay a certain fee, download the game and can start playing.

  - **Pros**: This is the simplest model and players are already familiar with it.
  - **Cons**: The model only provides one-time revenue and brings in less users than other models.
  - **Example**: Angry Birds, Cut the Rope, Tiny Wings.

- **Advertisement**: Displaying advertisements in a game is another popular revenue model. Popular advertisement networks include: iAd and Google AdMob.

  - **Pros**: Users do not have to pay for a game.
  - **Cons**: The model requires a large user base to become profitable and it amount to less money spent per user.
  - **Example**: Subway Surfer, Angry Birds Free.

- **Subscription**: Users must pay a monthly subscription to be able to play.

  - **Pros**: Recurring payments.
  - **Cons**: The game needs to be updated frequently to retain users and keep them engaged.
  - **Example**: Order & Chaos.

- **In-app purchase**:This is one of the most successful models for mobile games. Games are usually free to play and players have the possibility of purchasing virtual goods that can accelerate their progress and improve their experience.

  - **Pros**: Users themselves decide how much they want to spend.
  - **Cons**: Here, a good strategy is required to encourage users to spend money.

  - **Example**: Clash of Clans, Smurfs Village, Hay Day.

- **Advergames**: This model is used for games, made to advertise a specific brand or event.

  - **Pros**: These are simple games, and the failure of the game does not have a financial impact on the developer.
  - **Cons**: The success of the game does not have a financial impact on the developer.
  - **Example**: Pepsi Twist Shot, Sprite City.

Developers can decide to combine multiple models in their games (e.g. Death Really is a pay-to-play game with in-app purchases), or even release different versions of the same game, using different revenue models (Angry Birds is available either free with advertisements or pay-to-play).

7.2 Monetization in Run&Roll

Run&Roll falls into the category of endless runner games, which are considered to be easy-to-learn and easy-to-return-to, even after a longer period of abstinence. Users tend to play runner games while in states of waiting, like waiting in a line, taking a bus, etc. This type of game is highly suited for banners or interstitial advertisements, without being overly disruptive to the game experience.
We combined the advertisement and the in-app purchase models which enable users to play the game for free, but at the same time giving them the option of buying extra amounts of in-game currency.

7.2.1 Interstitial Ads

An interstitial advertisement is a full page advertisement that appears before a certain content page. One of the providers of interstitial advertisements is

Google's AdMobs – the one we chose for our game. Its implementation is not time consuming and can be seen in the following lines of code:

```
- (void)initTheInter
{
    interstitial_ = [[GADInterstitial alloc] init];
    interstitial_.delegate = self;
    interstitial_.adUnitID = ADMOBS_UNIT_ID;
   [interstitial_ loadRequest:[GADRequest request]];
}

- (void)runTheInter
{
    self.adsHaveBeenShown = YES;
    [interstitial_ presentFromRootViewController:self];
}
```

The *initTheInter* method initializes the interstitial advertisements with a unique ID, acquired when registering on the AdMob web page, and issues a request for an advertisement. When we want to show the advertisement, we call the *runTheInter* method which shows a previously requested advertisement. In Run&Roll, we display the advertisements before the start of each game session.

7.2.2 Run&Roll Shop

The Run&Roll shop is where users can spend their coins for goods that enhance their gaming experience and enable them to achieve higher scores. The shop is divided into two separate sections:

- "Boost/Utility" section: This is where users can upgrade their boost levels and buy extra utilities. We have also added a premium purchasable utility which doubles the amount of collected coins.

- "Get more coins" section: Here, users can acquire more coins, either by

purchasing one of our coin packages, or by performing certain actions that are rewarded (more on this topic in chapter 8).

Apple provides developers with the classes needed for implementing inApp purchases. All the products need to be created on the iTunes webpage, where the developer must select the appropriate product category. There are two categories of inApp purchases available:

- Consumables: Consumables can be bought and used more than once. They include items such as extra lives, in-game currency, temporary power-ups, etc.
- Non-Consumables: These are items and perks purchased only once and provide a permament effect. They include extra levels, unlockable content, etc. For non-consumable purchases, the restore purchases function is needed.

```
- (void)requestProductsFromAppleStore
{
    NSMutableSet * productIdentifiers = [[[NSMutableSet alloc] init] autorelease];
    NSArray * inAppProducts = [GameRepository getCoinPackages];

    for (int i = 0; i < [inAppProducts count]; i++) {
        [productIdentifiers addObject:[[inAppProducts objectAtIndex:i] objectForKey:
    @"productId"]];
    }


    SKProductsRequest * productsRequest = [[SKProductsRequest alloc]
    initWithProductIdentifiers:productIdentifiers];
    productsRequest.delegate = self;
    [productsRequest start];
}
- (void)purchaseProduct:(NSString *)productId onCompleteBlock: (id) block
{
    [self showProgressIndicator];
    //get the purchased product from the array of purchased products
```

```
    for(int i = 0; i < [self.products count]; i++){
        SKProduct * product = ((SKProduct*)[self.products objectAtIndex:i]);
        if ([productId isEqualToString:product.productIdentifier]) {
            self.purchasedProduct = [self.products objectAtIndex:i];
        }
    }

    if(self.purchasedProduct != nil){
        SKPayment *payment = [SKPayment paymentWithProduct:self.
    purchasedProduct];
        [[SKPaymentQueue defaultQueue] addPayment:payment];

        self.onPurchaseCompleteBlock = block;
    }
}
```

The *requestProductsFromAppleStore* function retrieves a list of products and displays it in the game store. The second function, *purchaseProduct:onCompletionBlock* checks if the product being purchased exists and makes the necessary security verifications, before finally starting the monetary transaction.

Level 8

To Arms My Friends: Socialization

In this chapter, we will discuss how to make a game social through social media integration. This aspect of Run&Roll includes integration of social networks (Facebook, Twitter, and YouTube), sharing, and rewarding players for following the game on aforementioned platforms.

Apple has been integrating social networks into their operating systems since the release of iOS 5.0, which makes application integration simpler. Since iOS 6.0, integration of Twitter, Facebook and Sina Weibo (the most popular social network in China) is provided. Instead of writing complex code, integration can be accomplished by importing the right frameworks into a project and calling the appropriate functions. Two social features were implemented into Run&Roll:

- Score sharing.
- Run&Roll subscription on social media sites (YouTube, Facebook, and Twitter): The subscription system rewards players for subscribing by awarding them coins for their support (Figure 8.1).

Score sharing was implemented by using the following code:

```
UIImage *img = [UIImage imageNamed:@"shareImage.png"];
```

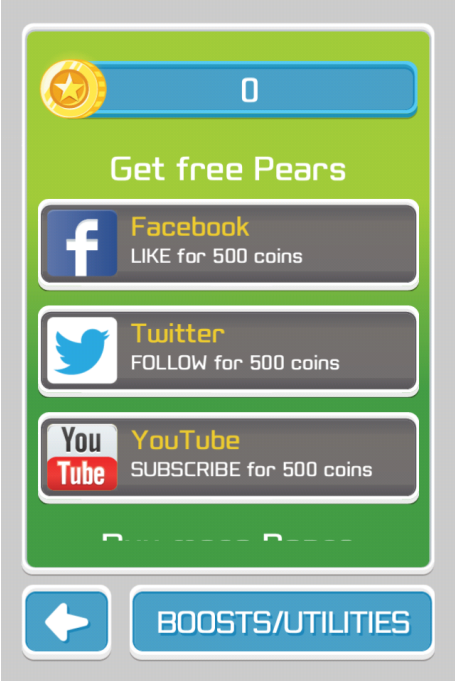


Figure 8.1: Shop Scene - Get More Coins

```
NSArray *shareArray = @[img, [NSString stringWithFormat:@"I just achieved %i
 points in Run&Roll! Try to beat me if u can! http://runandroll.net", score]];
UIActivityViewController *shareViewController = [[UIActivityViewController alloc]
 initWithActivityItems:shareArray applicationActivities:nil ];

[ self  presentViewController:shareViewController animated:YES completion:nil];
```

Run&Roll enables sharing results over social networks, such as Twitter and Facebook, as well as over SMS or e-mail (this can be seen in figure 8.2). Here, our goal was to implement a mechanism that will improve our game's viral effect, increase our player base and encourage users to play more, beat their friends, and finally bring in more users, without increasing the customer acquisition costs.

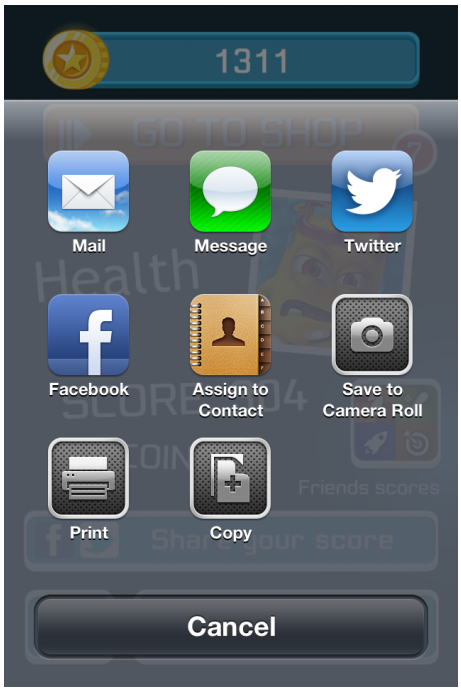


Figure 8.2: Game Over Scene - Share

Level 9

The Boss Fight: Competition

Every game developer tries to entertain and challenge players in his or her own (unique) way. Some design a detailed story line or breath taking graphics, others use realistic mechanics to make their game more appealing. Despite their differences, one thing they all have in common is the struggle to survive the market. And what a struggle this is! In the last couple of years, the number of newly released games on the iOS platform has risen to a staggering 300 games a day. In this stream of games, indie game developers have to find their own strengths and advantages, and hope that the game will appeal to the community.

What can a small indie company with a limited budget do to become noticed? I have already shared some of my thoughts on this subject in the previous chapter, where we discussed socialization. Further information on this topic will be given in the following subchapter.

9.1 Tracking user behavior

For tracking purposes, we use Flurry SDK, which is a build-measure-advertise-monetize framework that provides strong statistics mechanisms for customer tracking. Flurry SDK can track simple one-time events and complex multi

parameter time events. Using the administrator's console, events can be connected into funnels that provide detailed information on user paths, as seen in figure 9.1.

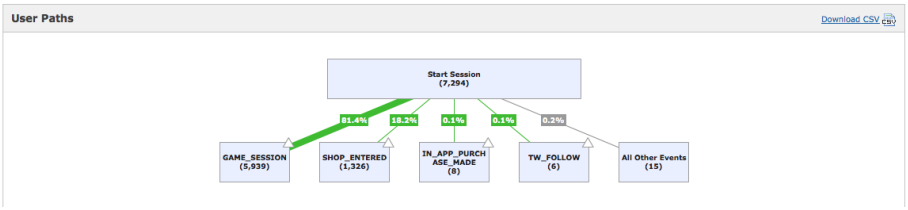


Figure 9.1: Flurry - user paths

In the following code snippet, two event registration calls can be seen. The first is a simple call which occurs when users enter the Run&Roll shop, and the second marks a more complex timed event which provides us with the duration of a game session, distance achieved, number of collected coins and the reason for the game ending.

```
// simple event log
    [Flurry logEvent:@"SHOP_ENTERED"];

// complex timed event log
    // send Flurry the length of game & the reason of death
    NSMutableDictionary *dictionary = [[[NSMutableDictionary alloc] init] autorelease
    ];
    [dictionary setValue: self .deathReason forKey:@"DEATH_REASON"];
    [dictionary setValue:[NSNumber numberWithFloat:[self.overLayer
    returnGameMeters]] forKey:@"DISTANCE_REACHED"];
    [dictionary setValue:[NSNumber numberWithFloat:[self.gameCoinsInstance
    returnGameCoins]] forKey:@"COINS_COLLECTED"];

    [Flurry endTimedEvent:@"GAME_SESSION" withParameters:dictionary];
```

9.2 Forums, Webpages, Communities

Another possible push comes from within the indie game communities which have grown strong and numerous and consist of early adopters, gamers, developers, reviewers, artists, and other gaming enthusiasts. They can provide us with (free) reviews, give related feedback, and help promote newly released game.

Epilogue: The Story Continues: After App Submission

After months of developing, improving, redesigning and fixing the game; when the game is finally in a state, where most people involved are happy with the product, the AppStore submission can finally occur. There, the Apple gurus will analyse the game, check for security breaches and if all goes to plan, give their blessings and allow the release of the game. However, questions still remain. Will the game be accepted? Will the user base grow large enough to cover the development costs or even bring in a profit?
Only time will tell.

List of Figures

1 Smartphones Manufacgturers Share by Operating System (Source:Silicon Valley Insider) 8

1.1 Apples IDE - Xcode 12

4.1 Hero - Sprite Sheet and a part of the coresponding plist file . . 30

6.1 Achievement triggered 41

8.1 Shop Scene - Get More Coins 50
8.2 Game Over Scene - Share 51

9.1 Flurry - user paths 54

Bibliography

[1] S. G. Kochan. *Programming in Objective-C (Developer's Library)*. Addison-Wesley Educational Publishers Inc, 2012.

[2] E. Sadun. *The Core iOS 6 Developer's Cookbook (4th Edition)*. Addison-Wesley Professional, 2012.

[3] R. Wenderlich, K. Hafizji. *iOS 6 By Tutorials: Volume 1*. CreateSpace Independent Publishing Platform, 2013.

[4] R. Wenderlich, A. Burkepile. *iOS 6 By Tutorials: Volume 2*. CreateSpace Independent Publishing Platform, 2013.

[5] R. Struogo, R. Wenderlich. *Learning Cocos2D*. Addison-Wesley Professional, 2011.

[6] S. Itterheim. *Learn iPhone and iPad Cocos2D Game Development: The Leading Framework for Building 2D Graphical and Interactive Applications*. aPress, 2011.

[7] I. Panberry. *Introduction to Game Physics with Box2D*. CRC Press, 2013.

[8] R. Hartson. *The UX Book: Process and Guidelines for Ensuring a Quality User Experience*. Morgan Kaufmann, 2012.

[9] R. Ford. *The App and Mobile Case Study Book*. Taschen GmbH, 2011.

[10] E. Ries. *The Lean Startup*. Penguin Books Limited, 2011.

[11] K. Werbach, D. Hunter. *For the Win: How Game Thinking Can Revolutionize Your Business*. Wharton Digital Press, 2012.

[12] S. Blank, B. Dorf. *The Startup Owner's Manual*. K & S Ranch, 2012.

[13] M. Geoffrey. *App Monarch: Secrets to building your own multi-million dollar apps*. AppNetworx, 2013.

[14] E. Catto *Box2D v2.2.0 User Manual*
Retrieved from:
http://box2d.org/manual.pdf

[15] D. Laughlin (2013) *Love, Courtship and the Promiscuous Male Mobile Gamer*
Retrieved from:
http://blog.flurry.com/?Tag=Monetization

[16] T. Paiva (2012) *Mobile Games Revenue Models*
Retrieved from:
http://www.slideshare.net/ThiagoPaiva/games-revenue-models

[17] Vergo Staff (2013) *iOS: A visual history*
Retrieved from:
http://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01915 / 2013
Datum: 2.4.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROK ČREŠNIK**

Naslov: **RAZVOJ NEODVISNIH VIDEO IGER**

INDIE GAME DEVELOPMENT

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Neodvisne video igre imajo naziv Indie igre, ki jih razvijajo posamezniki ali manjše skupine. V okviru diplome proučite koncepte takšnih iger, orodja za njihov razvoj ter obstoječe programske okvirje za razvoj takšnih iger. Na podlagi tega predstavite razvojni proces takšnih iger in se pri tem fokusirajte na organizacijske probleme razvoja v skupinah. Na koncu še razvijte primer igre na iOS Platformi ter predstavite in analizirajte probleme, ki so se pojavili pri razvoju.

Mentor:
doc. dr. Rok Rupnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
UNIVERZA V LJUBLJANI
2