

Keijo Sipola

ANDENGINE-BASED GAME FOR ANDROID

ANDENGINE-BASED GAME FOR ANDROID

Keijo Sipola
Master's Thesis
Fall 2015
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author: Keijo Sipola

Title of thesis: AndEngine-based game for Android

Supervisor: Kari Laitinen

Term and year when the thesis was submitted: Fall 2015

Pages: 55

The aim of this Master's thesis was to design, implement and release a game called seVer: The Hack and Slash for Google's Android mobile platform. The development team consisted of myself and Mikko Kemppainen. The expectation was to create revenue with the finished product and it was considered we would establish a company after making enough revenue.

The work was carried out in our spare time from 2012 to 2014 using Eclipse IDE with Google's Android SDK as the main platform with Java programming language. The game engine used was an open-source engine called AndEngine. The aim was to create unique gameplay experience combining hack 'n' slash elements to a side-scrolling viewpoint.

The game was developed to almost completion and a beta version was released for people interested in testing it and providing feedback and bug reports. The game did not get finished because of AndEngine's performance issues and final lack of motivation. The decision to stop working on the game was mutual and the gained experience has since been used in our later projects.

Keywords: Android, AndEngine, programming, Java, game, engine

FOREWORD

Being a gamer for 25 years, I'm extremely happy about the current state of tools available for starting game developers. The best game experiences I have had in last two years have been from indie games made by single developers. I was never going to get involved in more serious game programming until Mikko Kemppainen asked me to join their team for the programming contest and since then we have been starting a new project almost every year.

Therefore, my biggest thanks go to Mikko Kemppainen as our game seVer would have never seen daylight if we had not teamed up. It was great time even it had its ups and downs. I also want to thank my wife Maria for her support as there were lot of times I got stuck in front of the computer developing and yet we did not have to make any compromises. Also, my thanks go to Matti Urhonen for motivating me to start working on this thesis by getting his degree. My thanks also go to Kari Laitinen and Kaija Posio at Oulu University of Applied Sciences for supervising the thesis.



Keijo Sipola
29.11.2015
Reston, Virginia
USA

TABLE OF CONTENTS

ABSTRACT	3
FOREWORD	4
TABLE OF CONTENTS	5
TERMS AND ABBREVIATIONS	6
1 INTRODUCTION	7
2 SEVER: THE HACK AND SLASH	9
3 GAME ENGINES	14
3.1 Brief history of game engines	14
3.2 AndEngine	15
3.3 Other engines for mobile platforms	18
3.4 Box2D	18
4 THE DEVELOPMENT OF SEVER	19
4.1 Evolution of ideas and features	19
4.2 Programmatic overview of seVer	25
4.3 Hill algorithm	29
4.3.1 Algorithm explained	29
4.3.2 Segments to physics objects	33
4.3.3 Coloring the segments with meshes	33
4.4 Obstacles	36
4.4.1 Engine performance	37
4.5 Tools	38
4.5.1 RUBE	38
4.5.2 TexturePacker	40
5 MAKING PROFIT WITH THE ANDROID SOFTWARE	43
5.1 Google Developer Account	43
5.2 Ads	44
5.2.1 Banners	45
5.2.2 Interstitials	45
5.2.3 In-App purchases	48
5.3 seVer business model	48
6 CONCLUSION AND FUTURE PLANS	52
REFERENCES	54

TERMS AND ABBREVIATIONS

Term	Description
API	Application Program Interface
FPS	First-person shooter
XML	Extensible Markup Language
ARPG	Action Role Playing Game
OpenGL	Open Graphics Language
JNI	Java Native Interface
GPU	Graphics Processing Unit
Gameplay	How a game is played distinct from graphics and sound
Affix	An item attribute that increases the wearer's performance
Mesh	An OpenGL entity combining vertexes into draw instructions for the GPU

1 INTRODUCTION

This Master's thesis will introduce and explain a game development project on the Android platform from the initial idea to the implementation and beta release. The focus will be more on the design aspect and general overview instead of going into the details of the source code with the exception of an algorithm used in the game.

Mobile platforms are nowadays important gaming platforms. A mobile platform is an interesting platform to potentially create games with a small team and possibly even with a small effort, thanks to the modern game engines. Finnish mobile game industry has showed us many success stories including Angry Birds, Clash of Clans and Hill Climb Racing.

I did this project together with my old classmate and friend, Mikko Kemppainen. Our target was to create an addicting game which would combine gameplay elements from multiple genres, mostly from ones we love ourselves. We chose Android as the platform as we both had mobile phones running the Android operating system to test with and we saw the potential in Android's increasing market share.

The process consisted of discussing all the ideas we both had, choosing a game engine and starting to prototype all the chosen gameplay elements together to see if it would be a successful combination. We also wanted to separate our tasks so that we could both focus on the areas which were our strengths.

Modern game engines make developing games a breeze compared to the effort required 20 years ago. The most popular game engines also support multiple platforms where the software can be built for different platforms just by configuring a few settings and making minor platform specific tweaks in the program code.

Android's Google Play platform includes multiple APIs (Application Program Interface) which help to maintain the created projects, advertisement, updates, beta releases, track users and bug reports. In my opinion, this level of automation

at Google's end makes releasing the game much more tempting as this area of a game release is usually the one that slows down the project.

The project was carried out from 2012 to 2014 using our spare time therefore a typical work amount was around 10 to 20 hours per week for each of us. We also took a longer 6-month break from development during the start of year 2013 to gain some of the enthusiasm back as we struggled with the game engine during that period. These obstacles were pushed aside later and the game reached 90% completion in 2014. At this point we released an open beta version but we failed to finalize the game for an actual release. The open beta version of our game, seVer: The Hack and Slash, can be obtained by joining our Google+ group. [21]

2 SEVER: THE HACK AND SLASH

I have already created a few small games in my history. While they were never intended for a public release or even thought to have any users, there has always been a dream to release a game to the public and establish a user-base. The first time we tried to achieve this was in 2010 when I teamed up with my friends Mikko Kemppainen and Asko Eronen. From this cooperation came the idea to compete in a Finnish game programming contest where teams were competing to create the best game in 2 months. My role was to implement and produce dynamically changing music to a game where the player must destroy houses as fast as possible. I have some experience creating music with computers, therefore I was the natural choice for the role. It was a successful project which provided us the third place in the contest and a lot of experience in programming.

In early 2012, we decided to take on a new game development project. We started throwing ideas based on what kinds of games were currently popular on marketplaces and what kinds of games we enjoyed ourselves. For both of us, the biggest inspiration was Diablo game series. Created by Blizzard, they have refined the whole hack 'n' slash genre where the target is to complete quests fighting massive hordes of enemies while developing your character with abilities and picking up randomly generated gear as a reward.



FIGURE 1. Diablo 3 gameplay [1]

To take a completely new approach to the genre which is usually displayed from an isometric view (Figure 1), we decided to try to make the genre work from side-scrolling viewpoint. This also allowed us to make the gameplay simpler for handheld devices as a real hack 'n' slash game usually requires a complicated control scheme for different actions which is not possible on mobile devices, at least with fast-paced gameplay. The initial requirement was to have only two actions available on the touch screen: a hit button and the rest of the screen would register as a button for speeding up the character.

The inspiration for side-scrolling gameplay came from Tiny Wings, a mobile game created by Andreas Illiger (Figure 2). It was well-received by critics and users when it arrived for Apple's mobile platform iOS on 2011. I personally enjoyed the game myself a lot and we also competed for high scores between my friends. The game gives the player the control of a bird that cannot fly but can jump great distances by speeding down hills. The gameplay focuses purely on a single input where the user presses the screen to make the bird slide down hills. Releasing the screen lets the physics simulation to take care of the rest. Proper timing increases the momentum so mastering the mechanism provides a very satisfying gameplay experience. [2]



FIGURE 2. *Tiny Wings* by Andreas Illiger [2]

From this inspiration, *seVer: The Hack and Slash* was born (Figure 3). As mentioned before, the gameplay is inspired by *Diablo* series and *Tiny Wings*. The gothic theme has also been influenced by *Castlevania* game series. While the game does not have any major plot, the background is the hero who is an undead vampire wanting to revenge monsters. The hero starts his hunting runs at night and has only certain amount of time to progress until the dawn scorches him to death. This creates a repeating gameplay mechanism where the hero must obtain experience and better gear to make it to the next chapter before the sun rises.

Each chapter acts as a checkpoint from which the hero can resume. The monsters also increase in power in each chapter so sometimes it is necessary to keep on running the first chapters for better items before the new chapters come possible to survive. Also the loot quality the killed monsters drop increases by each chapter.

The game is controlled using 4 touch areas on the screen of the mobile device. The first button works mostly when pressed during downhill to make the hero to gain more speed. This mechanism is pretty similar to the one used in *Tiny Wings*. The hero keeps moving at a minimum speed all the time but that is not enough to pass any of the chapters. The second button causes the hero to jump to avoid

enemies, small hills or mud pits which slow the hero down. The jump button also acts as a flying button when the hero is airborne and he can glide to gain maximum advantage of the jump. The third is the hit button which causes the hero to hit with a charging mechanism where good timing will increase the damage of each hit. The fourth input is the rest of the screen and pressing that area picks up any items that are near the hero.

The items that are dropped consist of weapons, armor pieces, rings and trinkets. The damage is obviously the most important affix in the weapon as well as the attack speed. Different weapon types have different charging times and reach to suit different play-styles. Armor affixes are mostly related to protection, boot affixes affect speed and jumping power and rings and trinkets can have any affixes.

All the dropped gold is picked up automatically. The gold can be used to purchase, repair and enchant items though only enchanting was implemented. The idea for purchasing random items came from Diablo 3 as we enjoy the gambling nature of it. You can never know if you are getting the best item in game or the worst therefore it would have been definitely implemented at some point.

All of the gameplay elements were designed to create entertaining, addictive and repetitive gameplay in a good way. In my opinion, solid mechanics with repetition and progression can easily be better than a mediocre gameplay with a lot of different content.



FIGURE 3. SeVer main banner graphics

3 GAME ENGINES

3.1 Brief history of game engines

To understand the steps we took in our project, it is necessary to briefly go through the evolution and definition of game engines. It is fair to say that creating video games has changed drastically from the times of the first games, such as Pac-Man, to the blockbuster hits, like Call of Duty, which people are playing today. Modern games are nowadays built completely with game engines, either commercially available ones or the ones that the studios create themselves.

The evolution of game engines started in 1993 with id Software's Doom, a groundbreaking FPS (first-person shooter) video game for the PC platform. The reason for this was that the game was architected well separating the different components such as a core engine, a graphics rendering system and an audio system. Doom was a massive hit and is considered to be the pioneer for FPS games. As the game was programmed with the architecture that represented partially modern game engines, it created a chance for other studios to create games using the same engine. This was an optimal situation for both parties as the development group was able to create the games with less effort and the game would still be published by id Software to improve the time versus revenue ratio of both companies. [3]

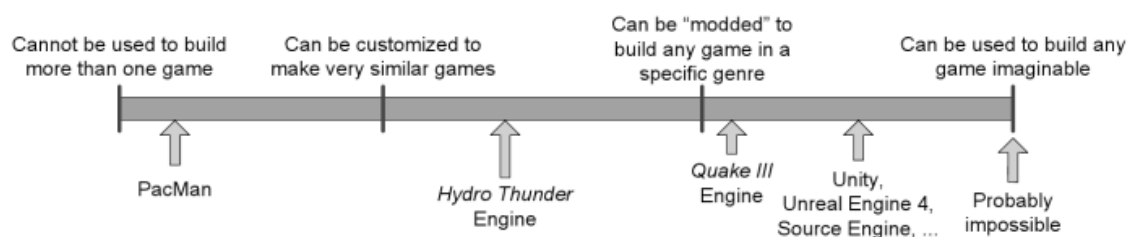


FIGURE 4. Game engine reusability gamut [3]

The idea of game engines is to allow users to have the core-work done for them already so that they can focus their development on the actual game instead of all technical details that run behind it. Even with key software components already

done, majority of the projects still have to incorporate a lot of custom software engineering. [3]

Most game engines used in modern games are still created for that specific genre even they are designed with reusability in mind. As Doom was a first-person shooter, it could not be used for a two dimensional platformer game without extreme modifications to the engine. Therefore it is relevant to distinguish game engines in that sense.

There are “in-house engines” that are built by a studio itself for a specific project such as a car racing simulation game. This engine is most likely to be used in their other racing games afterwards or they could even license it to other developers specializing in that same game genre. While this engine would be reusable, it would not make sense to modify it to run games from different genres. Quoting Jason Gregory: “It’s safe to say that the more general-purpose a game engine or middleware component is, the less optimal it is for running a particular game on a particular platform.” [3]

The other modern variant of game engines falls in the category where they are suitable for almost all genres. Most of the engines can even provide both 2D and 3D graphics capability. Also, the hardware of modern game consoles is so similar to PC hardware that it allows many game engines such as CryEngine and Unity to be used to build games for game consoles, too. [3]

A simple breakdown of a game engine is that it provides at least four basic components: game loop (core engine), graphics rendering, resource management and audio. While these can be split into multiple sub-categories, these components are usually the ones that the developer will be using and considering them as separate entities.

3.2 AndEngine

AndEngine is an open-source 2D game engine created by Nicolas Gramlich. It uses OpenGL for graphics and provides a variety of tools to develop games without understanding OpenGL and how game loops should be built. AndEngine is available through Github with some different versions. [4]

Although it might not be the most popular game engine for Android, we chose it to our game project as it allows fast development as of the easy framework. Most of the problems during the development were caused by the lack of documentation which we overcame by looking into the source code or by asking for help on the developer forum, which was fortunately quite active at the time of development. The referenced book “AndEngine for Android Game Development Cookbook” was not available at the time we started development but as we were quite confident that we already knew the basics of happening behind the engine, we could figure out everything ourselves. For the most of the time, this was correct as AndEngine is truly a user-friendly engine and mostly works logically.

As the concept and basics of game engines in general have now already been explained, we can have a detailed look how a life cycle of AndEngine game from creation, to minimization or destruction (Figure 5).

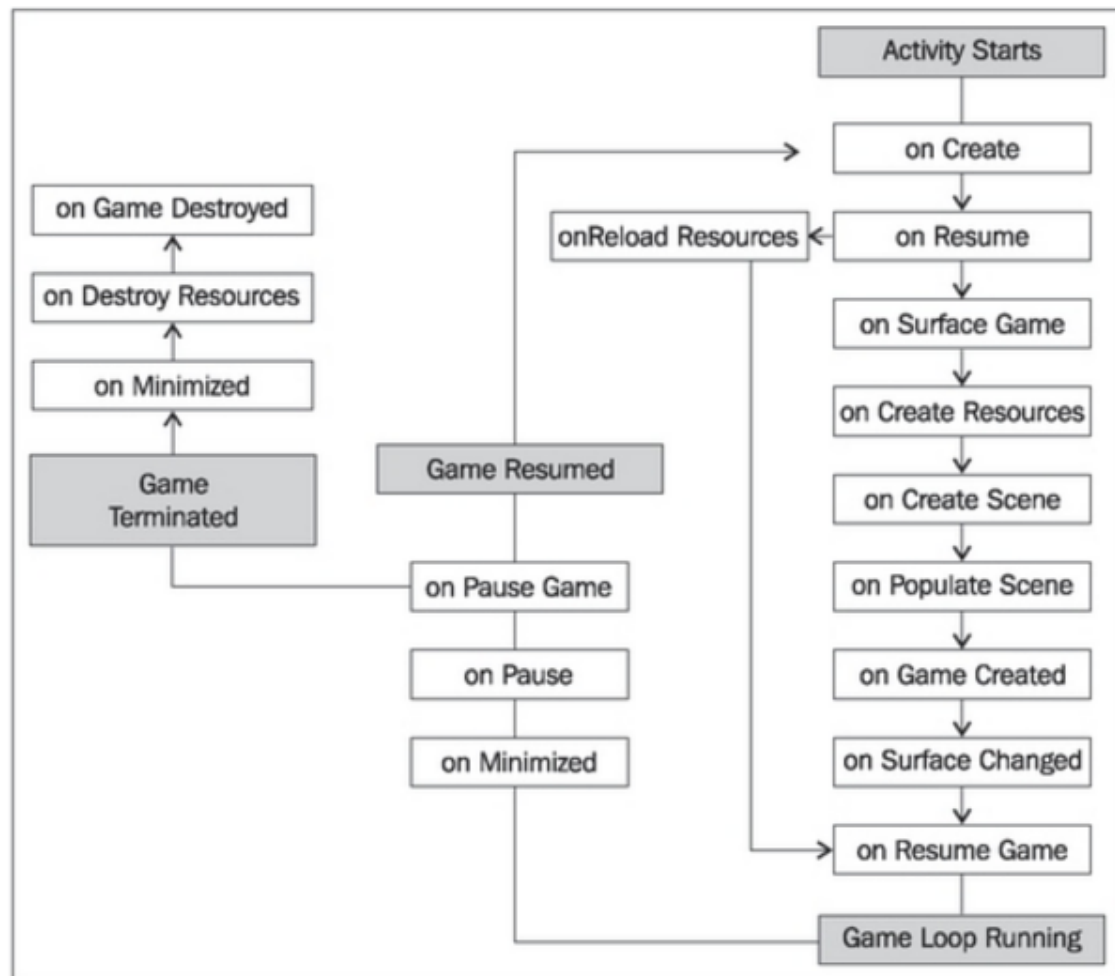


FIGURE 5. AndEngine game life cycle [5]

As anticipated, we ran into situations where certain functions or features did not work as we thought and this is where the value of open-source comes in. We were easily able to step inside all of the functions and see the developer's logic behind the scenes and see what worked as we thought and what did not. We even made some changes to the core of the engine ourselves to improve stability in the way we were using the engine.

Unfortunately, the engine is not being kept updated anymore by its developer Nicolas Gramlich as he has started his career at Zynga. There are still some slightly updated repositories on Github, which have been updated by most active users of the engine. Personally, I can still recommend AndEngine for small Android game development projects, especially for developers who already understand Java and want to try to make games.

3.3 Other engines for mobile platforms

For intermediate and advanced users, I would recommend Libgdx, which is a great open-source game development framework. The strongest features of Libgdx are its excellent performance, active community, vast number of tutorials and cross-platform support. The only downside is that Libgdx requires more initial programming and understanding to have a game running. The homepage of Libgdx says: “Libgdx is a Java game development framework that provides a unified API that works across all supported platforms”. They don’t consider it to be a traditional game engine for its modular structure where the developer can decide what features he wants to use. Developers can develop and run their games as PC Java applications and build them directly for Android platform with only slight modifications. Even Microsoft’s Windows Mobile and Apple’s iOS platforms are available as build targets so this is the best choice for creating multi-platform games. [6]

There are also engines which do not require that much knowledge in programming and rely more on scripting such as Unity, which is yet an extremely popular game engine and has been used in hit games such as Temple Run’s Android version, a game created by three people, which has achieved over billion downloads on mobile platforms.

3.4 Box2D

Box2D is an open-source physics simulation engine written in C++. It is made by Erin Catto and it is easily the most used 2D physics engine used on any platform today as it has been used in hit games such as Angry Birds and Limbo. It has been ported to many other programming languages as well including Java and C#. Box2D uses rigid body simulation meaning the bodies will never deform from their original shape. [7]

While there is a Java port available, AndEngine uses Box2D through Java Native Interface (JNI) which allows the use of libraries written in another programming language such as C++. The Java wrapper which is used in AndEngine and which takes care of the JNI calls, is actually originally provided by Libgdx. [4]

4 THE DEVELOPMENT OF SEVER

4.1 Evolution of ideas and features

The reality in game development is that the game changes drastically from the initial design, especially in an agile project. Prototyping is an important step in the beginning to see if the initial design has any potential or flaws. Usually, in a prototyping stage the graphics will mostly be placeholders drawn to support the gameplay. We have noticed in our previous projects that this can also affect negatively where the gameplay can be good, it will not have the same feel with the placeholder graphics and the idea will be ditched.

As we were well aware of this fact, we decided on day one that Mikko will be taking care of the graphics right from the beginning as he has been drawing his entire life and I will concentrate on the programming. We are both quite similarly qualified in programming but as the graphics play such a big role in a successful game, we decided to share our focuses and Mikko was welcome to join the programming side when the graphics were mostly finished.



FIGURE 6. SeVer development snapshot, a week in development.

Figure 6 shows a snapshot of our first prototype of the game. At this point we were going to implement the game with colored graphics. The main character

had two physics objects jointed together: a ball body which can move along the curvy ground relying on physics simulation by Box2D, and a box body, which was mainly there to give a better center of mass and tie the graphics to the object. The white curved line was based on the un-optimized hill algorithm, which used flat rectangles to display the position of the bodies that formed the hills. As a result, the performance was quite bad but for a working prototype, this was an excellent result, which we achieved in around one week. The background graphics were sample images provided by AndEngine. We used them to implement looping moving parallax backgrounds, which will be explained later.



FIGURE 7. SeVer gameplay snapshot, two weeks in development

The snapshot in Figure 7 was taken from a version around two weeks since the start of development. At this point we had implemented a zombie to test enemies and the main character had animated graphics and weapon swing animation. Additionally, the hair was tied to its own physics objects, which allowed some hair movement to get a sense of speed. Also, the arm holding the weapon had a dynamic physics object. This version had also the initially designed input methods where one button controls hitting with sword and touching the rest of the screen causes the player to “dive” for increased speed. We did not come up with any specific challenges at this point of development and all new features were coming along quite easily.



FIGURE 8. SeVer gameplay snapshot, three months in development

After three months of development, a lot of changes occurred as can be seen in Figure 8. The biggest change was that we gave up the colored graphics and went with black, gray and white colors. This silhouette styled look allowed Mikko to focus more on the animation and the figure of objects instead of the detail, which increased the graphics production time greatly. The overall look of the game also improved, according to our opinion. The version in Figure 8 also showcases the finished moving parallax backgrounds which gave each level a unique look.

The concept of parallax background includes multiple layers of background graphics. To gain a sensation of depth and movement in gameplay, each of the layers is moved at different speed to the opposite direction of the player. Using our later implementation and Figure 8 as an example, the one furthest back is a static image containing sky. The next layer is the distant mountains which will move with a very slow speed in relation to the player. The next layer contains smaller mountains with details and lighter color which move faster than the mountains on the previous layer but slower than the player. When the player and the foremost hills move with a higher speed, it gives the image of visual depth and a very good sense of movement.



FIGURE 9. SeVer gameplay snapshot, final version

Figure 9 shows the final version of the developed game. This is the version that was developed for almost two years. During those years there were some long periods where we did not have time to develop it at all. In comparison to the snapshot in Figure 8, there is a huge amount of changes and most of them cannot be seen from a screenshot.

The gameplay evolved to have more complexity as we introduced jumping and flying. As the gameplay was designed to progress based on character's abilities, we added a stamina attribute to affect the gameplay the most. The player has to control when to hit, jump, fly or accelerate as all of these reduce the stamina which is gradually restored based on the equipped skills and gear. We also changed the hitting mechanism so that the player has to hold and release the hit button at the right time to make the most damage to the enemies.

The screenshot in Figure 9 also shows the ragdoll mechanism on the enemy and physics-based blood particles. We introduced complex physics driven ragdolls for the dead enemies to make them look more dynamic. The tools used for this are explained in a later chapter.

A huge portion of the time went to developing all the elements outside the actual gameplay. The inventory system seen in Figure 10 is one of the most important views in the game as the player will go through all the found items after each run. While the inventory looks fairly simple, a lot of logic happens in the background such as calculating how the new gear would affect current stats and displaying that to the player. Previously our implementation included a grid inventory where different types of items took different amount of space from the grid. While it was more realistic, managing that kind of inventory with a screen of a mobile device turned out to be more difficult than useful.



FIGURE 10. SeVer inventory screen

The inventory screen also provides access to the enchant screen (Figure 11). The enchanting is an idea borrowed from Diablo series where the player can use gold to make an item more powerful. The basic enchanting follows the flow where normal items (white) can be enchanted to magic items (blue) with more stats. The magic items can be enchanted further to rare items (yellow) with additional stats and possibly better base stats. All of the items affixes can also be rerolled in the enchanting screen. Affixes are additional stats that affect player's attributes such as "Extra damage". Rerolling will replace the selected affix with another random affix which will make the item potentially better or even worse. For example, if one affix is "Experience given x%" and the user does not need to gain experience

faster, he can try to reroll it to i.e. “Speed x%” or “Curse x%”. We also added an additional tier of unique “uber” items, which are very rare drops from killed enemies. Uber items are a lot more powerful than the regular ones and have also a unique look and graphics. Uber items cannot be enchanted from magic items but they can be rerolled.

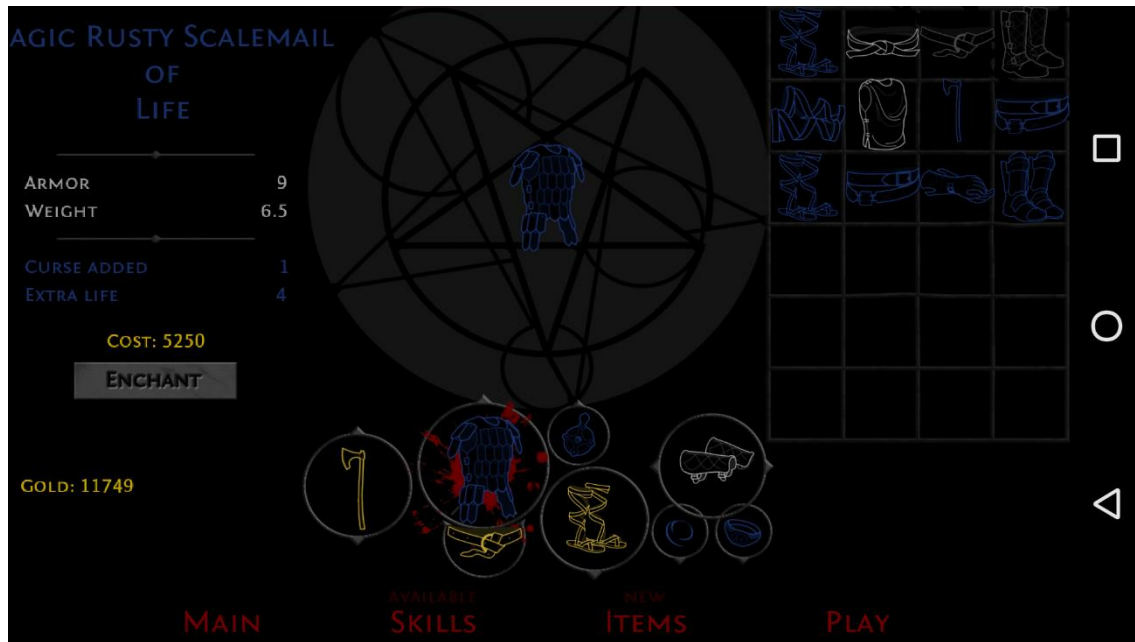


FIGURE 11. Item enchanting screen

Another late addition to the game was the skill system shown in Figure 12. The current set of skills provides only passive effects and they mostly affect the attributes and stats of the hero. The idea was to also introduce active skills such as ranged magic attacks or temporary invisibility which the user has to trigger but we did not have time to design how this could be implemented on UI without adding too much new buttons.

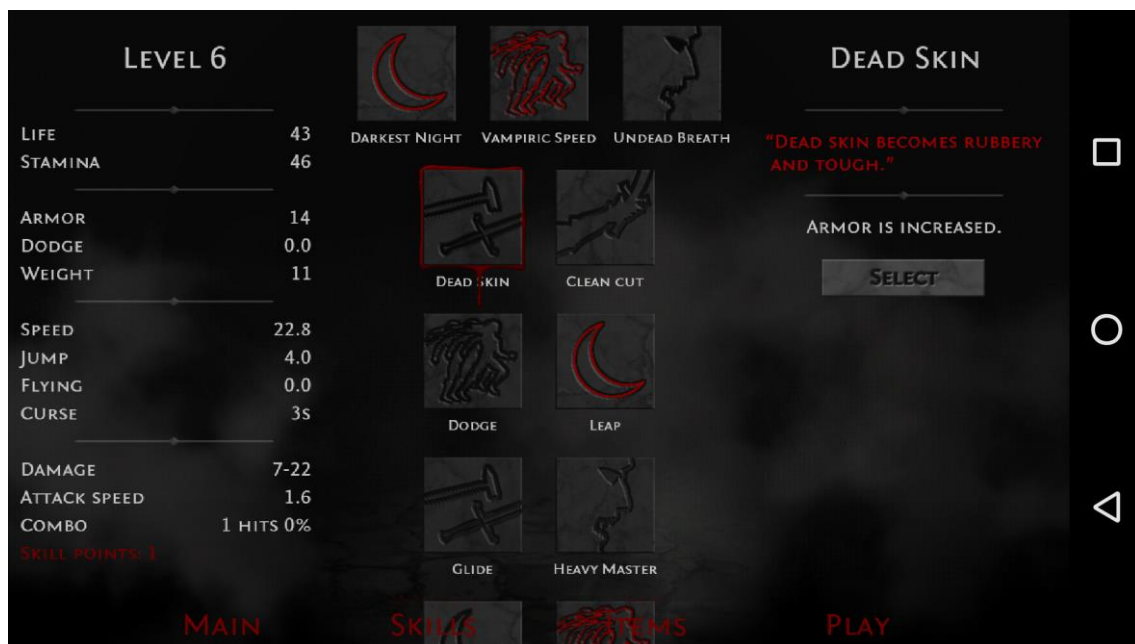


FIGURE 12. SeVer skill selection screen

4.2 Programmatic overview of seVer

To get a better picture of all the implementation required for the game, see Figure 13. Those 17 packages contain all the classes we wrote on top of the game engine. The packages are a technique in Java to divide the classes into logical groups and give them a namespace. This provides the modular programming and access protection for the classes. The packages can also be compressed into JAR files to be used in other projects more easily.

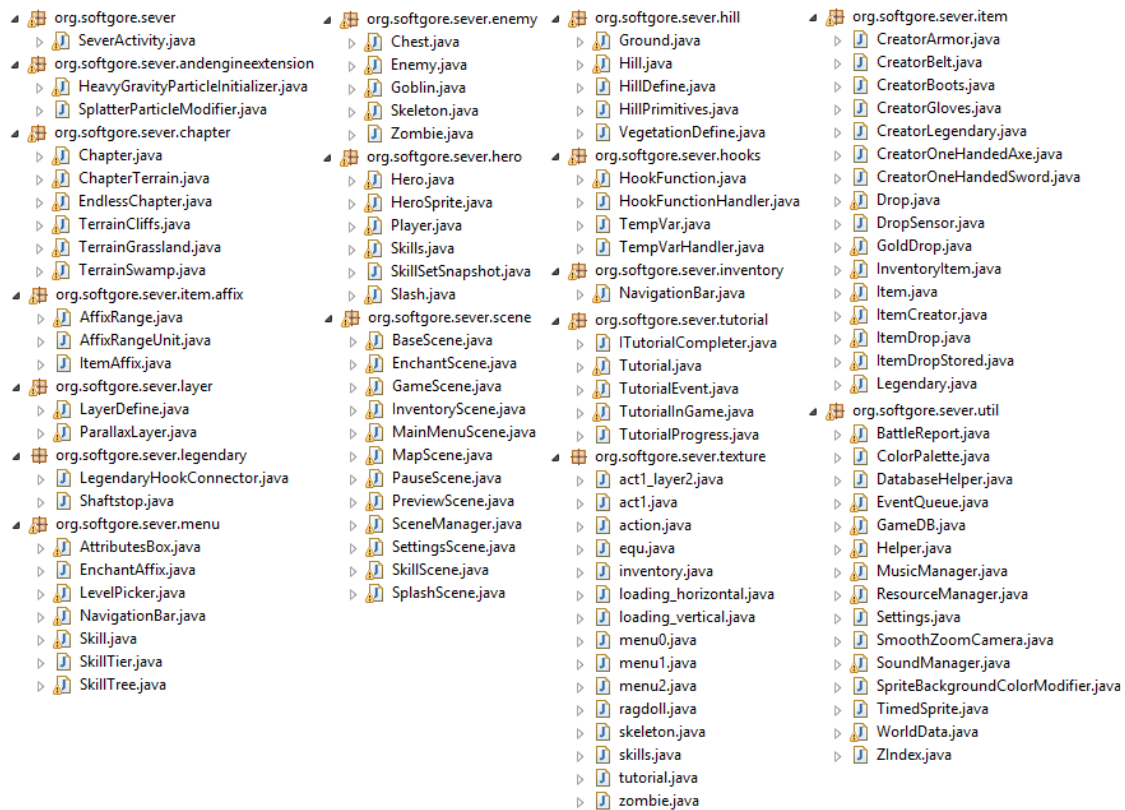


FIGURE 13. The classes written for seVer

In the package “sever”, we have only our main class, “SeverActivity”. This class implements the most important class of AndEngine, BaseGameActivity. This class has all the required implementation for the game engine to start running over the original Android activity and to display any graphics the user adds via the engine.

The “chapter” package implements the logic for the levels. The chapters are loaded from XML files and they contain all the information we need to keep the game loading gameplay content dynamically while the user is playing the game. This information includes all the hill definitions including relative positions, graphics for the hills, graphics for the parallax background graphics, enemy types and obstacles such as mud pits. We have also made a basic implementation for endless chapters as a proof of concept with gradually tougher enemies, steeper hills and better rewards. The endless chapters are generated completely randomly so that gameplay experience would be unique each time. We could also store the seed used for the random functions and use the best ones we come

across with for a concept of daily runs. In a daily run, we would use the same seed for all players for one day at a time so they could compete against each other to gain the best result in daily leaderboards.

The “enemy” package has our Enemy superclass which all the individual subclasses inherit. The Enemy class has all the sprites, physics objects and position information stored. This allows us to initialize some enemy specific objects within subclasses but we can use the main functions to control and dispose the enemies easily. The unique functionality in subclasses includes picking the right graphics, the initial position of physics objects and definition of ragdoll objects for death animation.

This architecture also allowed us to easily implement chests in the game. The chests are basically static enemies which do not collide with the player but use the hit detection and item dropping mechanics from the Enemy class. Even the opening animation is based on the ragdoll functionality we use for animating enemy death.

The “hero” package contains many classes to implement and define all the mechanics for the main character in the game. One especially useful class we created is HeroSprite class. It takes advantage of the AnimatedSprite class of AndEngine which allows an automated changing of the tiles of sprites at certain time intervals. In my experience, this is the most usual way to create simple animated graphics in most 2D game engines. As smooth dynamic animation is a crucial part in seVer to make it look good, we created a heavily overwritten and customized version of AnimatedSprite. The class can change the tile changing speed dynamically based on the outer input, which in this case is the speed of the hero’s main physics object, and also swap to the jumping frame without skipping frames.

The “scene” package is one of the biggest packages in the game code-wise. The scene can be described as the canvas where all the entities are tied to. Usually, only one scene is displayed at a time and it gives a good way to separate different views in the game such as the title screen, different menus and the actual gameplay view. Our biggest scene is the MainScene, which is a class that ties all

the gameplay objects and mechanisms together. The most called function in Scene is “onManagedUpdate(float pSecondsElapsed)” which will update all the entities attached to the scene. All the custom logic and calls that need to be done on each frame goes inside this function, such as triggering the creation of hills based on the camera position and updating hero's animation in the function in MainScene. The physics engine will also step in relation to the pSecondsElapsed so that the simulation stays on par with the graphics.

The “tutorial” package is one of the latest additions to the program. As we gave the application to a small group of people for initial testing, we soon realized they did not know how to play it and we had to build a tutorial. The concept is that we detect if the game is launched for the first time. This will start a series of dedicated levels with instructions and detection so that the user takes the right actions.

Items have one of the most important roles in the game and have so many features implemented for them that we had to create a dedicated package to keep them organized. Items are heavily influenced by the existing action roleplaying games (ARPG) as for their attributes, naming conventions and categories. We built versatile item generator classes which will create items with random properties based on the type of the drop. In ARPG games, the names of the items are as important as the affixes. It is much nicer to find a sword called “The Holy Protector of The Weak” than “Greatsword”. A code snippet of a dynamic item name prefixing can be seen in Figure 14.

```

private String getMagicItemNamePrefix(ItemAffix affix) {
    String prefix = "Magic";
    switch (affix.mAffixType) {
        case Item.ITEM_AFFIX_MODIFIER_DAMAGE:
        case Item.ITEM_AFFIX_VALUE_DAMAGE:
        case Item.ITEM_AFFIX_VALUE_EXTRA_DAMAGE:
            if(affix.mValue>10.0f) {
                prefix = "Cruel";
            }
            else {
                prefix = "Sharp";
            }
            break;

        case Item.ITEM_AFFIX_VALUE_WEIGHT:
        case Item.ITEM_AFFIX_VALUE_EXTRA_WEIGHT:
            if(affix.mValue < 0f) {
                prefix = "Light";
            }
            else {
                prefix = "Heavy";
            }
            break;

        case Item.ITEM_AFFIX_VALUE_ARMOR:
        case Item.ITEM_AFFIX_VALUE_EXTRA_ARMOR:
            prefix = "Armored";
            break;
    }
}

```

FIGURE 14. Dynamic item name prefixing based on main affix

4.3 Hill algorithm

As the gameplay relies heavily on physics simulation on the hills, this chapter will explain the creation of the hills in more detail. The hills are generated from XML files which define each chapter. The coordinate system is a 2-dimensional system where each hill has a coordinate for the middle point and the ending point.

4.3.1 Algorithm explained

Each hill begins from the ending point of the previous hill with the exception that the first hill starts from origin (0.0). These points are then used in an algorithm which calculates the points for the physics objects to create curvy profile to the hills.

The algorithm is based on the cosine function which can be configured with one parameter to create less or more smooth hills. This parameter will be referred as a segment width.

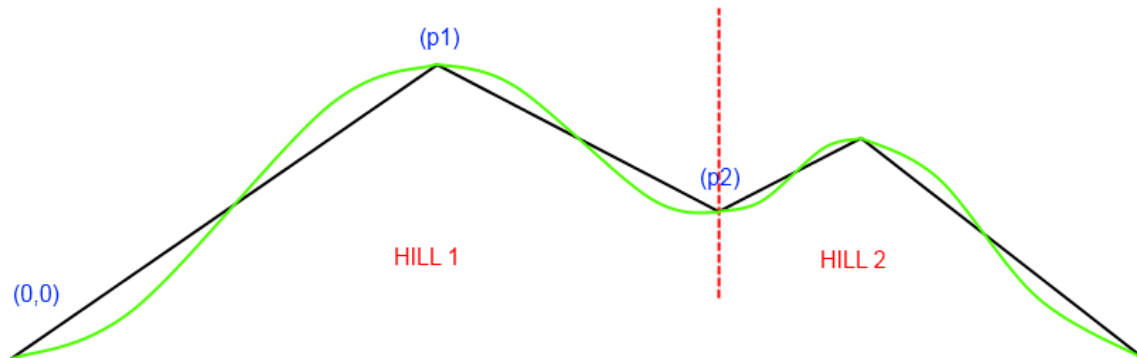


FIGURE 15. Creating hills between fixed points

As stated before, the hills have three points that define the key points for the hill, which can be seen in Figure 15 marked with blue. The first hill has the origin as p_0 . Taking a look into creating the first curve between p_0 and p_1 , we first have to calculate how many segments the curve will be divided into. This is calculated by dividing the difference between the x-coordinates of p_0 and p_1 with the defined segment width. The number of segments affects the performance heavily as a physics object will be created for each segment.

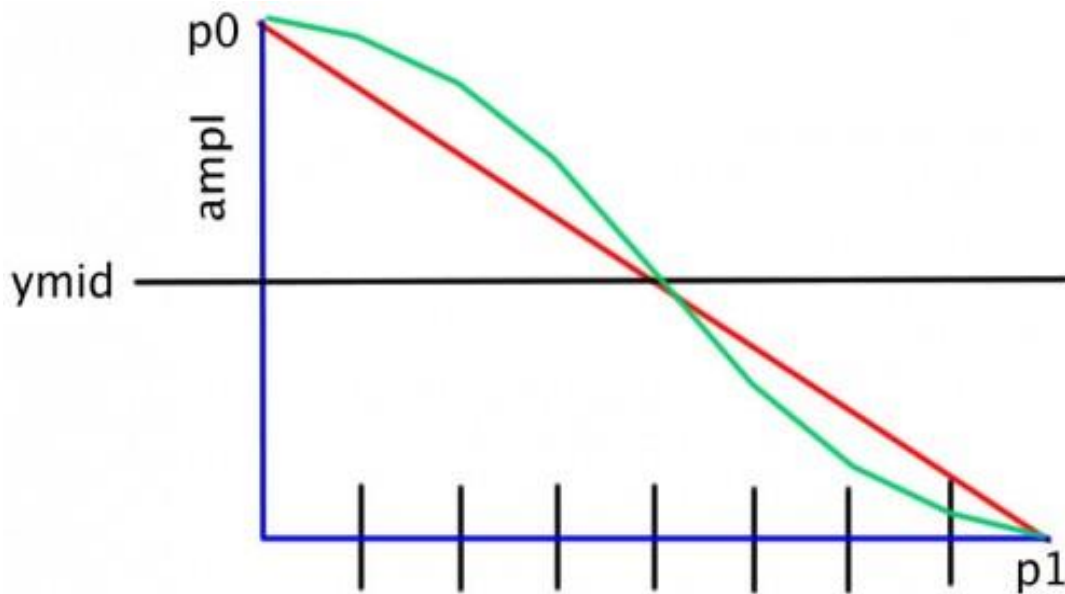


FIGURE 16. Curve divided into 8 segments [8]

We also need to define the width of each segment by dividing the difference between x-coordinates of p_0 and p_1 with the number of segments we got from the earlier calculation. Another key variable is the delta angle which is calculated by dividing π with the number of segments. Also, the middle point between y-coordinates ($ymid$) and the vertical difference ($ampl$), which are self-explaining (Figure 16) need to be calculated.

The hill is created with two iterations as one hill consists of 2 cosine curves. In Figure 16 we can see the latter piece of the hill which would be calculated in the second iteration.

```

Vector<Vector2> hillPoints = new Vector<Vector2>();
float segmentWidth = 10;

for (int i = 1; i < keypoints.length; i++) {
    Vector2 p0 = keypoints[i - 1];
    Vector2 p1 = keypoints[i];

    int hSegments = (int) Math.floor((p1.x - p0.x) / segmentWidth);
    float dx = (p1.x - p0.x) / hSegments;
    float da = (float) (Math.PI / hSegments);
    float ymid = (p0.y + p1.y) / 2;
    float ampl = (p0.y - p1.y) / 2;

    Vector2 pt0 = Vector2Pool.obtain();
    Vector2 pt1 = Vector2Pool.obtain();

    for (int j = 0; j < hSegments + 1; ++j) {
        pt1.x = p0.x + j * dx;
        pt1.y = (float) (ymid + ampl * Math.cos(da * j));
        pt0 = pt1.cpy();
        hillPoints.add(pt0.cpy());
    }

    Vector2Pool.recycle(pt0);
    Vector2Pool.recycle(pt1);
}

createMeshHill(hillPoints, def);

```

FIGURE 17. The algorithm in Java

Figure 17 shows a Java code example of the algorithm where the previously explained key variables are calculated in the first parts of the loop (dx, da, ymid, ampl). The explanation of the code will be done using the second iteration as it follows the visualization better (Figure 16). The actual segment positions are calculated in the inner for-loop of the algorithm. For each segment, the x-position is calculated by multiplying the current segment index with the delta x and adding it to the position of p0.x, which is the x-coordinate of the peak of the hill. [8]

In the next line, calculating the y-position is the most important part of the algorithm as it takes advantage of the cosine function. By looking at the characteristics of a cosine curve in Figure 18, we can see that we have to map $\cos(0)$ at p0 and $\cos(\pi)$ at p1. We can then calculate the y-position of each segment with the Math.Cos() function with the help of ymid and ampl in the programming example in Figure 17. [8]

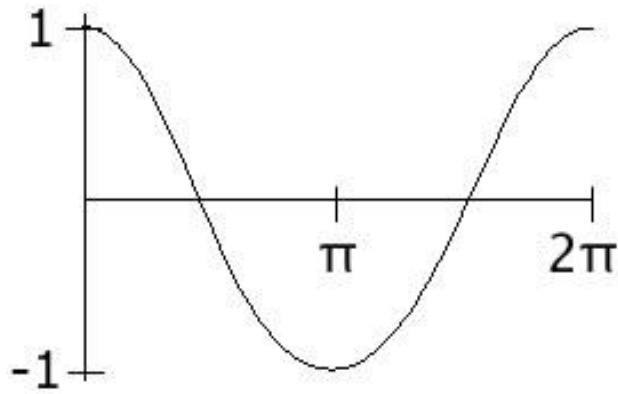


FIGURE 18. Cosine curve [8]

4.3.2 Segments to physics objects

The hill algorithm provides only the points for the hill profile which could easily be used to draw a line. But instead we have to create Box2D physics objects for gameplay purposes. Each segment will be presented in a separate physics object, Box2D line body. Line bodies are basically polygon shapes but derived into lines for a simpler usage.

AndEngine Box2D extension provides a helper class called PhysicsFactory to create physics bodies. The line bodies can easily be made with a for-loop that iterates through all the points that were created by the algorithm. The key parameters for the bodies are the starting point and the ending point of the line as well as the group index which is set so that the bodies will collide with the bodies of the player. This will allow the player to move on the created hills instead of falling through them.

4.3.3 Coloring the segments with meshes

The points the algorithm generates are also used to color the hills. The color scheme of seVer is mostly black and white therefore the coloring is quite simple task as we do not need to apply any textures or use different colors for different parts of the hills.

Coloring is done by using OpenGL meshes, which are sets of vertices that define the points, colors and possible texture coordinates to be drawn. Meshes are one

of the base entities in OpenGL, which makes them fast to draw as the instructions are handled by a “pipeline”. This conversion of objects into pixels by the OpenGL renderer communicates directly with the GPU (graphics processing unit) of the device. Basically, the mesh object in Figure 19 is internally telling the GPU to wait for a vertex array, passes the calculated array and tells how to draw the vertices and the GPU takes care of displaying the pixels. [9]

```
float colBlack = Color.BLACK_ABGR_PACKED_FLOAT;
float[] meshData = new float[hillPoints.size() * 6];

for (int i = 0; i < hillPoints.size(); i++) {
    Vector2 pt = hillPoints.elementAt(i);

    // MESH DATA
    // bottom point
    meshData[i * 6] = pt.x;
    meshData[i * 6 + 1] = pt.y - 1000;
    meshData[i * 6 + 2] = colBlack;
    // top point
    meshData[i * 6 + 3] = pt.x;
    meshData[i * 6 + 4] = pt.y;
    meshData[i * 6 + 5] = colBlack;
}

final Mesh meshGround = new Mesh(0, 0, meshData,
    hillPoints.size() * 2,
    org.andengine.entity.primitive.DrawMode.TRIANGLE_STRIP,
    mActivity.getVertexBufferObjectManager(), DrawType.STATIC);
```

FIGURE 19. Mesh creation in Java

The same points that are used to create the physics objects are used to create the meshes. The points are converted into vertices meaning that each vertex is given a position and a color value. For the start point of each segment, another vertex is required. This vertex will be positioned 1000 pixels lower as we need to make sure that the hill is colored in black even when the camera gets zoomed out in some occasions.

In the program example in Figure 19 we can see that first a float array is initialized with the number of generated hill points multiplied by 6. The multiplication is required as 3 values are needed for each vertex including the x-position, the y-

position and the color. The additional 3 are for the vertex that colors the mesh downwards.

In the programming example where the actual Mesh object is created, the most interesting parameter is the DrawMode which is defined as a TRIANGLE_STRIP. The draw mode is an instruction parameter which tells the GPU how all the vertices should be drawn in the mesh. The instruction handles how the vertices are connected, and the order in which the vertices are defined is important for different draw modes to work correctly. AndEngine supports only lines, triangles and a triangle strip. The triangle strip is the most suitable draw mode in seVer as it connects all the vertices in such a way that it creates a completely black terrain. Figure 20 shows the main draw modes supported in AndEngine.

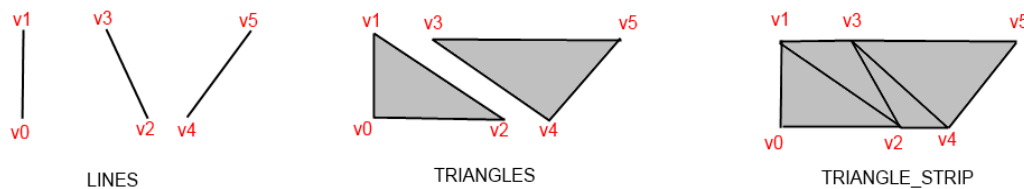


FIGURE 20. Basic OpenGL draw modes supported in AndEngine

As the game is developed for the mobile Android platform, a good memory handling and performance tweaks are a necessity. Each frame needs to draw four hills on average while some of the hills are as wide as the screen. It was decided that each hill consists of individual segments and all of them together share one mesh.

If we would do it in such a way that only each segment would be using its own mesh, this would actually increase draw calls. We also need the physics objects to exist off-screen as the loot is thrown forward and many times it lands outside the camera rectangle and the collision is detected with between the physics objects of the item and the ground.

The stable and memory efficient hill generation with fast rendering graphics is one the key points in seVer. This was achieved with two development iterations where the source code was reviewed and hierarchical changes were made. The

current state of the algorithm is at a sufficient level performance-wise and might gain only a slight advantage from a float arrays instead of Vectors. A general rule is that new objects should not be generated in the runtime of the game thus recycling objects is highly recommendable. This is why our implementation has a Vector2Pool which can be used to recycle Vector2 objects.

The meshes are in their final stage as the programming is done almost directly to the OpenGL interface. The evolution of seVer included the ideas of adding new features to the algorithm in the future as we designed a new set of chapters with a desert theme which will involve dunes. To achieve dunes instead of plain hills, the algorithm would need to be improved to support other kinds of curves.

4.4 Obstacles

Our aim was to make the game run at the resolution of 1980x1080 at 60 frames per second. At the time we made this decision, most high-end Android devices released had this as the native resolution. We were aware that our project might take a year or two therefore we decided to create all of our graphics for this resolution to make the game future-proof.

One way to deal with the high resolution graphics on older devices is to create additional lower resolution versions of all the graphics. When loading the resources, Android API allows to query the device for the supported resolutions which can be used to define if the game should use low or high definition graphics.

The graphics were made with Inkscape which is a drawing program that supports vector graphics. The idea behind vector graphics is that the user draws all the images and the tool takes care of storing the graphics in a vector format. This allows the graphics to be rendered later using any resolution and zoom level without losing any detail by the rasterization engine of the tool. This would allow storing the same images easily by saving them in low and high resolution formats. We realized this possibility but for the current state of the project, we decided to provide all the graphics in high definition. This does not make as notable performance hit as thought at first; the rendering resolution used will still be the highest that the device supports when the engine is initialized with EngineOption

containing FillResolutionPolicy. Only the texture memory load will be increased.
[10]

4.4.1 Engine performance

As our project processed onwards and more features requiring processing speed were implemented, we started to notice problems with the performance. We started to go through online forums to see if other people were facing similar problems as we did and to our concern, we saw a lot of advanced programmers having these problems, too.

While AndEngine struggles displaying multiple sprites, the graphics entities of the game, a big part of the performance problems in our game comes from the vast number of physics objects displayed and calculated on each displayed frame. To take an example, let's take a frame which displays 3 hills, 3 enemies and the player. With a horizontal resolution of 1080 using 10 pixels as the pixel step for each hill, we have always at least 108 pieces of static physics objects for hills calculated. Usually, this number is even more as we have the objects created further to support the spawning enemies and to capture dropping items. Even though the objects used for hills are static, which means the engine does not calculate any forces affecting them, they still need to calculate contacts with dynamic physics objects such as the objects in player's and enemies' bodies.

The player object consists of 6 dynamic physics objects. One enemy consists of 30 objects in average if we calculate in the ragdoll physics where each body part has their own physics object. This means one frame must be capable of calculating forces between over 200 physics objects, calculating the graphics that are attached to them, updating the positions and drawing the whole scenario including the parallax background and the vegetation on hills.

This scenario was very tough to render at 60 frames per second even with a modern smartphone with a quad-code processor, an advanced GPU and 2 gigabytes of RAM (Random Access Memory). AndEngine is simply not up to this kind of task where a lot of sprites need to be displayed at the same time. To achieve a playable experience with slower devices, we decided to include an option where the ragdoll physics will not be spawned and simple ghost image of

the enemy will be displayed upon destruction. Forcing the ragdolls would have made the game unplayable with any older phone and removing them also removes 30 sprites per enemy which is especially helpful with AndEngine. At that point that we noticed we cannot do much to improve the situation further, around 80% of the development was done. This was one reason for us to lose some of the motivation to finish the game but we still decided to try.

4.5 Tools

The IDE (Integrated Development Environment) we used in the project was Eclipse, which is the tool mostly used for Java development while it also supports other languages such as C++ and Python. The Android SDK integrates well with Eclipse. It was also the recommended IDE by AndEngine, which made it the obvious choice for us.

We used SVN (Apache Subversion) with the TortoiseSVN client for the source control which allowed us to keep track of all of the changes and to have the code up to date. The server hosting SVN, which has scheduled backups and secured access, is set up on our friend's server so that we have all our source code still stored.

4.5.1 RUBE

To create a complex ragdoll physics for dead enemies, defining the bodies and graphics programmatically was out of question because of the sheer number of objects per enemy. We came across with a Windows software made by a company iforce2d called R.U.B.E, Really Useful Box2D Editor, which will be referred to as RUBE in this chapter. [11]

The user can import their own graphics, which they use in the game, to RUBE and draw and define the physics objects they want to attach to the graphics. This is extremely helpful especially when the user wants to combine multiple physics objects together with different types of physics joints. The work can be exported into a JSON file (JavaScript Object Notation), which is a syntax for storing and exchanging data. [11, 12]

Most of the maintained 2D game engines that support Box2D have a RUBE JSON file support. Fortunately AndEngine also had this support created as an extension by an active user “nazgee”. We preload all the RUBE exports in the game when the resources are loaded in the startup and we also load all the graphics for them. As we create the ragdolls dynamically every time an enemy is killed, we wanted to make sure that we do not load any new graphic resources at that point. This is still a relatively slow process to do in the game as we create all the physics objects dynamically. For example, a zombie ragdoll consists of 32 separate images and physics objects (Figure 21), therefore there is room for optimization here so that all the physics objects would be recycled per each enemy type similarly to the Vector2 objects explained previously.

The biggest visual drawback for this kind of approach was that, as can be seen in Figure 21, the starting position of the ragdoll is always fixed to the same position. Each enemy has its set of animated frames therefore when an enemy is killed when the current frame is the most different to the fixture defined in RUBE, we see a small jump in frames which seems like an animation missing few frames. Luckily, the game is quite fast-paced thus this is not a big issue but it is something that could be considered to be implemented in a more complex way.

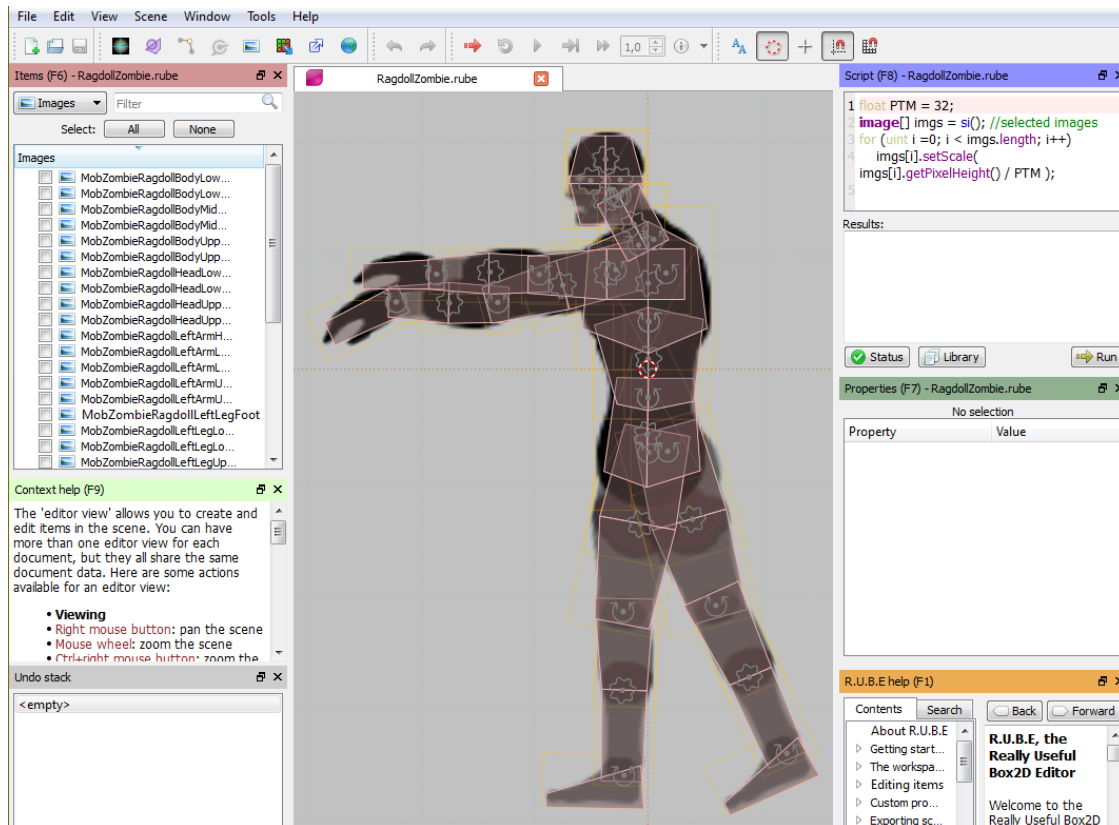


FIGURE 21. User-interface of RUBE with zombie ragdoll

4.5.2 TexturePacker

Another very useful tool we purchased was TexturePacker by CodeAndWeb. It is a Windows application that automates packing images into one big image which can then be exported and loaded in game engines including AndEngine. The export has a specific AndEngine option which exports the positions of individual images in the bigger image to an XML file in a format that AndEngine can parse. A partial sample export of our equipment graphics which are used in the game to display the collected equipment in the inventory screen can be seen in Figure 22. [13]

```
<?xml version="1.0" encoding="UTF-8"?>
<texture version="1" file="gfx/atlas_equ.png" type="bitmap" width="512" height="1024"
    pixelformat="RGBA_8888" minfilter="linear" magfilter="linear"
    wrapt="clamp" wraps="clamp" premultiplyalpha="false">
  <!-- Created with TexturePacker http://www.codeandweb.com/texturepacker -->
  <textureregion id="0" src="_000000_empty.png" x="502" y="141" width="1" height="1"
    rotated="false" trimmed="false" srcx="0" srcy="0" srcwidth="1" srcheight="1"/>
  <textureregion id="1" src="_000001_amu.png" x="246" y="808" width="64" height="64"
    rotated="false" trimmed="false" srcx="0" srcy="0" srcwidth="64" srcheight="64"/>
  <textureregion id="2" src="_000002_amu_1.png" x="466" y="737" width="41" height="56"
    rotated="false" trimmed="false" srcx="0" srcy="0" srcwidth="41" srcheight="56"/>
```

FIGURE 22. A partial sample texture export from TexturePacker

The images were drawn with a supportive coloring so we can call AndEngine's setColor() function for the entity. This will change to outline colors of the entity so that we can display different quality of items by using a single image per object. A part of the spritesheet which the export loads is displayed in Figure 20. Different coloring of some of these items can be seen in Figure 10.



FIGURE 23. Equipment spritesheet

To load all the textures from the XML files, we created an automated routine (Figure 21) that we call when we load all the resources at onCreateResources (Figure 5). In this way, whenever we add new graphics, we can just update the new images to the defined folders in TexturePacker, run the export again and update the xml files to the project. All the individual images can then be loaded as textures by requesting a texture by its name. For example, to get a texture for an amulet from the spritesheet, we would call function "ResourceManager.getInstance(). getTexture("_00000_amu");". The naming convention of the images follows specific rules so that we can iterate them easily and request a certain version of the texture based on the unified naming convention.

```

AssetManager am = getInstance().mActivity.getAssets();
try
{
    String[] xmlList = am.list("gfx/texture_xml");
    for(String xmlName : xmlList)
    {
        try
        {
            TexturePack texturePack = new TexturePackLoader(getInstance().mActivity.getAssets(), tm).load(
                getInstance().mActivity.getAssets().open("gfx/texture_xml/" + xmlName), "");
            texturePack.loadTexture();
            String shortName = xmlName.substring(0, (xmlName.lastIndexOf(".")));
            getInstance().mTextureLibraries.put(shortName, texturePack.getTexturePackTextureRegionLibrary());
            sleep(30);
        }
        catch (final TexturePackParseException e)
        {
            Debug.e(e);
        }
    }
}
catch (IOException e)
{
    e.printStackTrace();
}

```

FIGURE 24. Texture loading routine in onCreateResources

5 MAKING PROFIT WITH THE ANDROID SOFTWARE

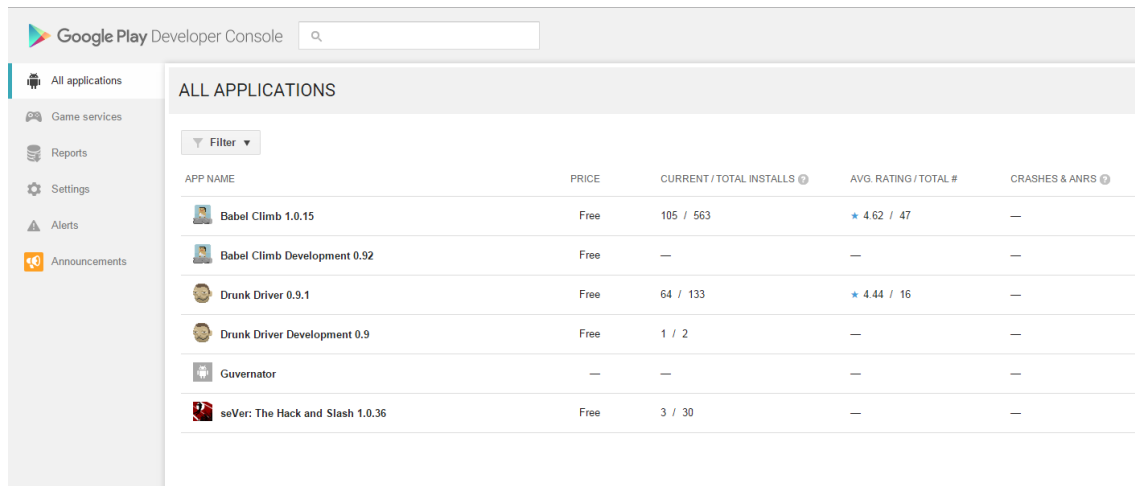
As Android is an open platform, anyone can program, build and create Android applications with Google's Android SDK without registering to any service. Anyone can install software which they have received for example via e-mail if they have a security setting called "Unknown sources" enabled, which is accessed through device settings. However, there are significant advantages using Google Play, Google's own application marketplace. Google Play incorporates a lot of features which make earning money a lot easier to the developer such as In-App billing and licensing which protects the priced apps to not to be downloaded illegally. The applications on the official marketplace also have less change to contain viruses or malware code. [14]

5.1 Google Developer Account

To use Google Play, a developer must create a Google developer account. Google will require a one-time payment of 25USD for each developer account created. As the developer account can be a shared account between multiple users, thus in our case we created only one account to be used. This is a relatively cheap price compared to for example Apple's marketplace, "App Store". They require a subscription which will cost the developer 99USD per year. [14, 15]

The Google developer account is accessed via a developer console web page. The main page gives an overview of all the applications added to the system. The overview also includes information of all current installs, total installs, ratings and crash reports (Figure 25). A developer can add multiple different applications and even multiple versions of the same application. The main reason for adding the same application multiple times is to add a developer version which can be tested for features which take advantage of Google's own services such as leaderboards and achievements API before releasing the application. We did not implement any Google API services for seVer as we did with our other later game projects therefore there was no need for separate developer version of the application (Figure 25). The main focus in seVer was to improve the gameplay as much as possible therefore we went only with an open beta release and our

idea was to focus on the leaderboards and achievements later, even after the initial release. [16]



The screenshot shows the Google Play Developer Console interface. On the left is a sidebar with navigation links: All applications, Game services, Reports, Settings, Alerts, and Announcements. The main area is titled 'ALL APPLICATIONS' and contains a table of applications. The table has columns for App Name, Price, Current / Total Installs, Avg. Rating / Total #, and Crashes & ANRS. The applications listed are Babel Climb 1.0.15, Babel Climb Development 0.92, Drunk Driver 0.9.1, Drunk Driver Development 0.9, Governorator, and seVer: The Hack and Slash 1.0.36.

APP NAME	PRICE	CURRENT / TOTAL INSTALLS	AVG. RATING / TOTAL #	CRASHES & ANRS
Babel Climb 1.0.15	Free	105 / 563	★ 4.62 / 47	—
Babel Climb Development 0.92	Free	—	—	—
Drunk Driver 0.9.1	Free	64 / 133	★ 4.44 / 16	—
Drunk Driver Development 0.9	Free	1 / 2	—	—
Governator	—	—	—	—
seVer: The Hack and Slash 1.0.36	Free	3 / 30	—	—

FIGURE 25. Google Developer Console

”Out of the 34 hours and 45 minutes users spend on mobile each month, 89% of that time is spent on apps.” It is clear there is revenue to me made with mobile applications. We were also aware of the high third-party application usage on the Android platform when we started our project. There are multiple ways to make money with your application even if you are providing your game initially for free. [17]

5.2 Ads

Most of the free applications on the market incorporate advertising. This has become increasingly more popular lately as there was not that many advertisement providers with programming interfaces available at the time Android was still starting to grow. While there are many providers available nowadays such as Millennial Media, Adfonic and Tapjoy, I will focus on Google’s own advertising solution called Admob.

AdMob started as a single man’s project in 2006 and Google realized its potential in earning money and acquired the company in 2010. AdMob is also available for iOS and Windows Mobile operating systems so that developers with cross-platform applications can focus their effort on one advertisement platform. [18]

We decided to choose AdMob as the advertisement provider for our project as it was provided by Google itself. In my opinion, this is a major reason why it is the most popular advertisement platform for Android as the Android SDK and tutorials are available on the same website as the Google Mobile Ads SDK which is used for AdMob. It gives developers more confidence as they do not have to go through the background of the advertisement company knowing it is a familiar one.

There are multiple types of advertisement one can incorporate into an application. In my experience, the most commonly used advertisement type is banners.

5.2.1 Banners

Banners are usually displayed in a fixed size slot of the screen, top or bottom, during certain conditions such as during the main screen or in “game over” screen (Figure 26). Banners can be chosen to take on multiple actions such as installing an app, taking a user to a certain website, providing directions or viewing products. The banner can also open a full screen advertisement, such as a video, showing more details about the item being advertised. The best part of banners is that they usually do not affect the user experience of the application in a negative way for being fixed in size. Most of the time, the banners contain animated images to engage the user. Developers can also set an option to promote their other own applications in the adverts. [17]



FIGURE 26. Smart banner example [19]

5.2.2 Interstitials

The other common advertisement type is called interstitials (Figure 27). They usually have the feel and look of a web application providing their own close button and navigation. Being a full display advertisement, interstitials are usually displayed during breaks, such as between levels, or when noticing that the user has been using the application for a while. Sometimes interstitials are used in

applications too often and appear in times when the user is about to make a selection on the application and are accidentally forced to make a selection on the advertisement, which is a bad practice. [17]

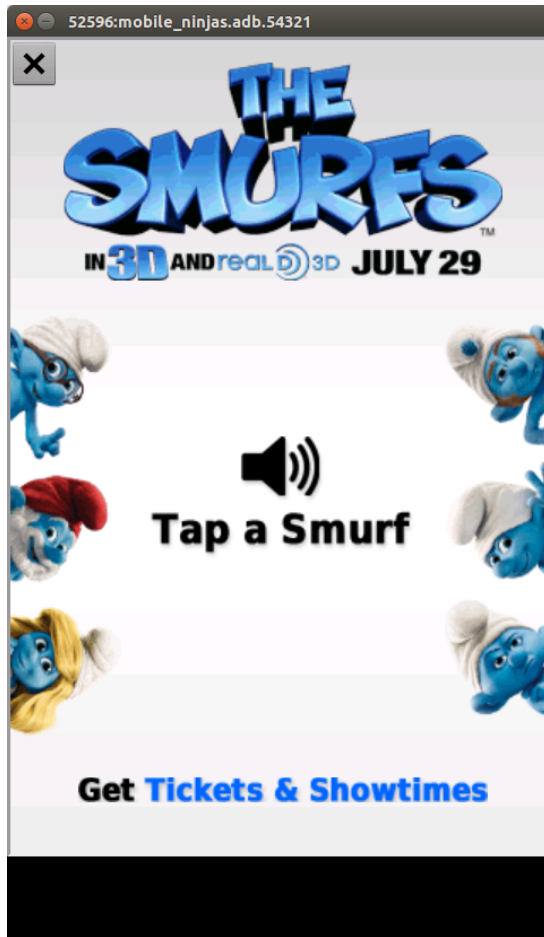


FIGURE 27. Interstitial ad [20]

Based on my experience, this is also a good point when designing an advertisement as there are many users that might uninstall the software if they feel that they are being directed too forcefully to click on ads. The recommended place to put interstitials is between levels or stages where there is a natural loading time in the application itself. Interstitials can be configured to be shown only to certain people and the developer can make that call within the app based on for example if the user has made In-App purchases which are explained later. [17]

AdMob implementation

Implementing a simple smart banner (Figure 26) within an Android application is really straightforward. The advantage of a smart banner is that it is automatically centered to the user's screen and uses a transparent background. To gain most advantage, the developer should check the user's display size when the application starts so that they choose the best height for the banner out of three options. [19]

Taking advantage of Google's SDK for AdMob, the developer does not have to configure anything related to their account and the actual loading process and they can simply use one AdView class (Figure 28).

```
AdView adView = new AdView(this);  
adView.setAdSize(AdSize.SMART_BANNER);
```

FIGURE 28. AdMob smart banner implementation [19]

If the code snippet from Figure 28 is used directly in Android's main activity and that takes care of drawing all the elements, it does not require more effort to display an advertisement. Based on my experience, the game engines provide their own drawing canvases which are built on top of the main activity so this would not work directly with most of the game engines. This requires acquiring the view object from the activity within the engine and drawing the ad at the right position, usually as the last object so it does not get overdrawn by the game engine's objects.

The developer can also implement an AdListener class from the SDK, providing them events such as when user clicks the application (Figure 29). This kind of information is useful as it can be used to trigger certain conditions within the software such as not displaying anymore ads for a while if the user has actually clicked on the ads. [19]

```

public abstract class AdListener {
    public void onAdLoaded();
    public void onAdFailedToLoad(int errorCode);
    public void onAdOpened();
    public void onAdClosed();
    public void onAdLeftApplication();
}

```

FIGURE 29. AdMob AdListener abstract class [19]

5.2.3 In-App purchases

The other commonly used business model within a free application is In-App purchases. In-App purchases are considered as virtual goods or subscriptions to be sold within the application. Usually, in games the purchases allow power-ups, a customization of your game character or a faster progression. In-App purchases are usually impulse buys and based on Google: “In fact, according to a recent study done by IHS, a world-class global research company, US respondents said that up to 89% of the money they spend on gaming apps was for in-app purchases.” [17]

The biggest benefit of having In-App purchases is to get users to download the application for free and if they are enjoying it, get revenue from the most dedicated users or the ones that want to progress fast. It is still good to remember that you need to grow your user-base to make a profit with In-App purchases as only minority of users still makes these purchases. The best practice is to utilize multiple monetization practices while thinking the end user to make the most revenue. [17]

5.3 seVer business model

While we did not make it to a commercial release, we had lots of ideas for the actual business model for seVer: The Hack and Slash. We decided to release our game for free to get an initial user-base and try to make money based on other means.

Our business model would incorporate ads and In-App purchases. One of the most common ways to allow users to get rid of an annoying advertisement is to give them the possibility to make an In-App purchase for virtual goods for a certain amount, which will also remove the advertisement. In my experience, this is the best way as I have come across with applications which provide a free and a priced app which require separate installs. Providing one install and virtual goods with the removal of ads is definitely appealing to the users.

To incorporate ads, our idea was to add a banner to the main screen and the “game over” screen. Our main screen is a perfect example to hold a banner in the top corner without violating the user experience as all of our control items such as links to other screens are at the bottom of the screen (Figure 30). We would also have shown banner ad always in the game over screen. We included interesting statistics of the finished run such as enemies killed, distance travelled and experience gained to the game over screen therefore placing a banner into a view that is going to be looked for a period of time was logical. We were also thinking of adding interstitial ads sometimes after game over but as we both had bad experiences of them, we did not consider them for the initial launch.

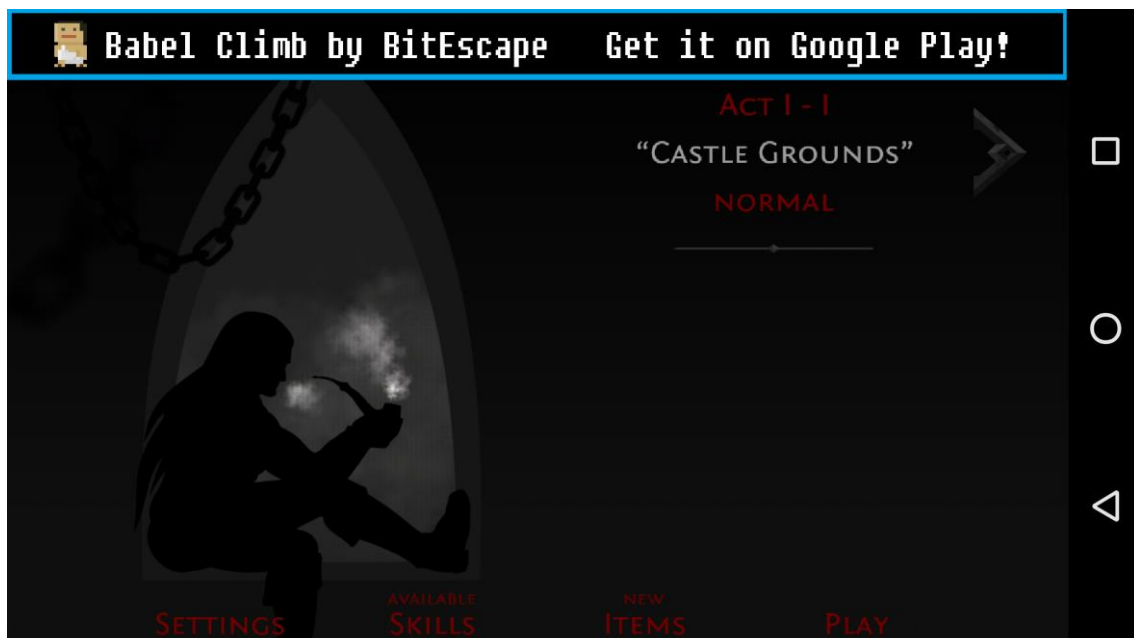


FIGURE 30. SeVer main menu with a banner ad on top

The idea to engage a player to start using In-App purchases in seVer was to make any In-App purchase to also remove the ads hence removing the necessity for a separate free and charged version of the application. We saw this approach used in FingerSoft's Hill Climb Racing and thought it was the best way to approach.

The first decision was to include an In-App purchase which allowed just supporting us, an indie development team, with a small amount of money and in exchange we would remove the advertisements and allow the user to choose a new color of smoke for the pipe tobacco the game's main character is smoking on the main screen (Figure 30). This would have been relatively simple to implement and we would have definitely got some initial revenue from our friends and co-workers.

As the player's progression throughout the game is heavily dependent on the gear he discovers while playing, we decided to build In-App purchases to support this kind of progression while forcing players still to play instead of just spending money to progress. As the player collects in game currency "gold" to perform tasks such as buying random items, repairing items and enchanting items to allow them to progress in the game or the leaderboards, there would be players who need more gold. The gold price to amount relation would have been non-linear where for one dollar, the player would get 50,000 gold, two dollars would get him 150,000 and five dollars would get him 600,000. With relatively small sums and large increases, it would increase the probability for the user to use more money and increase our revenue.

We had an idea to implement breakable items as users might use In-App purchases to purchase gold in game to repair all the items and allow them to continue to play with their favorite equipment. The third action requiring gold is item enchanting. As every reroll will cost more than the last one, it is another possible situation where the user might buy gold as an In-App purchase to afford the enchanting.

The other In-App purchase would have been a season pass, which would have given the user access to all the upcoming content we had planned for the game.

The progress in seVer is chapter based and the user would have paid for example \$5 to get an unlimited season pass so that they could access Chapter 2 as soon as we release it as an update. The free app users would have to stick to the designed 10 chapters in act 1 and compete in the randomly generated endless chapter with a level cap of 20. This level cap means that the hero cannot gain any new abilities or use higher level weapons. The season pass holders would have gotten a removed level cap so that they could go even over the level 20 and get the best scores on the “per-chapter-leaderboards” as their character has better stats. To tease the free application users we would have dropped items with a level requirement of over 20 which they could not equip to make them more interested buying the season pass to equip those items.

6 CONCLUSION AND FUTURE PLANS

The results of the project can be considered as a success as well as a failure. As the aim was to release the game on Google Play, it has to be considered as a failure as we got only to a beta phase and were not able to carry out the last effort to finish it. Most of our struggles came from the engine performance. Partially, we also started to lose belief in success before we let enough people to try the game. While we got some appraise from people who tried the beta, we also got a lot of bug reports and saw that people were uninstalling the game quite quickly after trying it. We knew about some of the problems in the game ourselves, such as the progression was not balanced. It sounded very tough task to take on as there are hundreds of variables we would have needed to rebalance.

If the results are viewed from a learning standpoint, I personally could not be happier with them. It was the first Android software development project for both of us and I honestly think it was more ambitious than majority of the small projects out there. We were able to create complex implementations and architecture and while it was refactoring after refactoring occasionally, there is no better way to learn programming than through your own mistakes. We already had the basic knowledge of how game engines work before we started the project, but seeing and understanding the performance issues and changing some of the functions in the core of the engine itself were an experience that will carry on in our professional careers in a positive way.

Although we have not discussed the future of the game for a while, we both agree that we created something we cannot just give up. The first step in reviving seVer would be to port it to the Libgdx engine, which I actually prototyped once during the development. Unfortunately, I soon discovered that the engines are so different that it would not be worth the effort as the game was around 70% of completion at that time. If I was to start working on seVer again, I would change the gameplay mechanics, too. That would give a fresh setup and keep myself interested in the project.

One way to finish the game could also be to just drop out some of the features of the game. I would personally like a more fast-paced gameplay thus getting rid of most of the progression through gear, the progression would come from learning to play the game, more like in Tiny Wings.

Even we abandoned seVer, we did not stop making games. We took all we learned, started to learn to program on Libgdx and released our first full game “Babel Climb” on Google Play Store in the end of 2014. It is a simple tower climbing game but it still carries a ton of what we learned from seVer in the background. Although it is not a massive hit now in the end of 2015, I am still very proud of what we have achieved. I am ready to start a new project already.

REFERENCES

1. Blizzard 2015. Diablo 3. Date of retrieval 17.11.2015.
<http://us.battle.net/d3/en/media/screenshots>
2. Illiger, A. 2011. Tiny Wings. Date of retrieval 17.11.2015.
<https://itunes.apple.com/us/app/tiny-wings/id417817520>
3. Gregory, J. 2014. Game Engine Architecture, Second Edition. Taylor and Francis Group.
4. Gramlich, N. 2013. AndEngine. Date of retrieval 17.11.2015.
<https://github.com/nicolasgramlich/AndEngine>
5. Schroeder, J. – Broyles, B. 2013. AndEngine for Android Game Development Cookbook. Packt Publishing.
6. BadLogicGames 2015. Libgdx. Date of retrieval 17.11.2015.
<https://libgdx.badlogicgames.com/features.html>
7. Catto, E. 2015. Box2D. Date of retrieval 17.11.2015. <http://box2d.org/about/>
8. Wenderlich, R. 2013. How to create a game like Tiny Wings. Date of retrieval 17.11.2015. <http://www.raywenderlich.com/32954/how-to-create-a-game-like-tiny-wings-with-cocos2d-2-x-part-1>
9. Noble, J. 2013. Basics of OpenGL. Date of retrieval 17.11.2015.
<http://openframeworks.cc/tutorials/graphics/opengl.html>
10. Inkscape 2015. Date of retrieval 17.11.2015.
<https://inkscape.org/en/learn/faq/>
11. Campbell, C. 2015. RUBE. Date of retrieval 17.11.2015.
<https://www.iforce2d.net/rube>
12. Refsnes Data 2015. JSON Tutorial. Date of retrieval 17.11.2015.
<http://www.w3schools.com/json/>

13. CodeAndWeb 2015. TexturePacker. Date of retrieval 17.11.2015.
<https://www.codeandweb.com/texturepacker>
14. Google 2015. Date of retrieval 17.11.2015.
<http://developer.android.com/distribute/tools/open-distribution.html>
15. Apple 2015. Date of retrieval 17.11.2015.
<https://developer.apple.com/programs/how-it-works/>
16. Google 2015. Developers Console Help. Date of retrieval 17.11.2015.
<https://developers.google.com/console/help/new/>
17. Google 2015. Making Money: The App Monetization Playbook. Date of retrieval 17.11.2015.
<https://services.google.com/fb/forms/admobmonetizationplaybook>
18. Google 2010. Date of retrieval 17.11.2015.
<https://googleblog.blogspot.com/2010/05/weve-officially-acquired-admob.html>
19. Google 2015. Date of retrieval 17.11.2015.
<https://developers.google.com/admob/android/banner?hl=en>
20. Google 2015. Interstitial Ad. Date of retrieval 17.11.2015.
<https://developers.google.com/mobile-ads-sdk/docs/dfp/android/interstitial>
21. Sipola, K. – Kemppainen M. 2014. seVer: The Hack and Slash beta. Date of retrieval 22.11.2015.
<https://plus.google.com/communities/106950395878658078990>