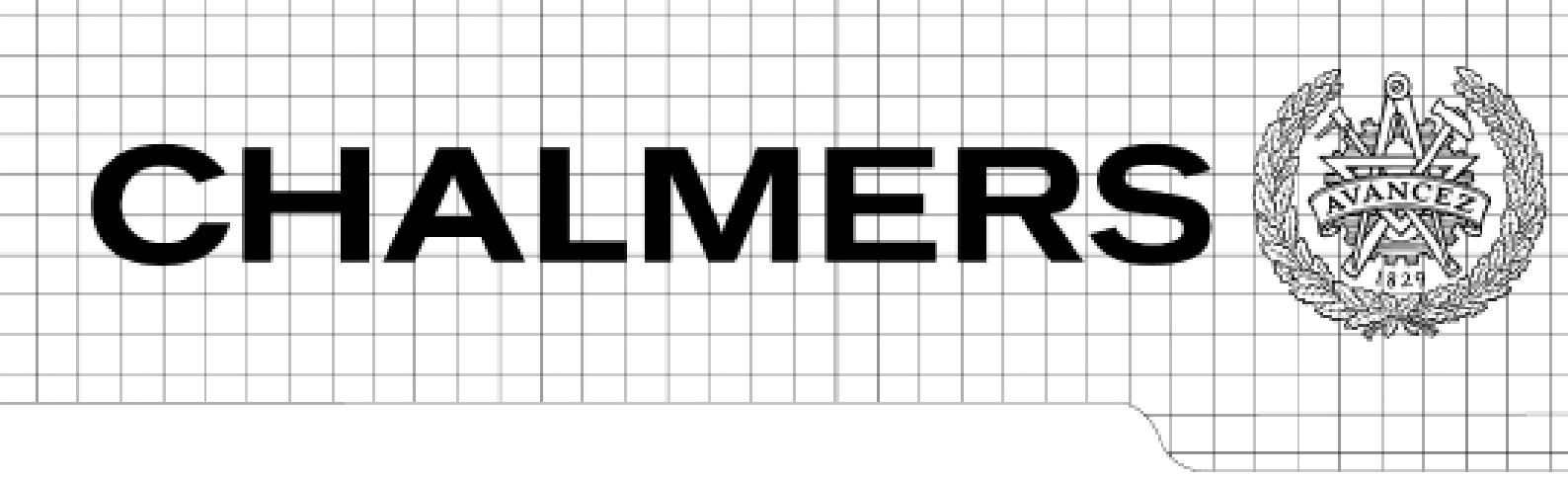

Tower Defense for Android

Bachelor's Thesis
Computer Science and Engineer Programme

Authors:

Jonas Andersson
Daniel Arvidsson
Ahmed Chaban
Disa Faith
Fredrik Persson
Jonas Wallander

Department of computer science and Engineering
Division of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden 2010

Abstract

This thesis focuses on game development for Android, an operating system for mobile phones developed by Google. The market for Android applications is currently growing rapidly, and there are many games currently available. The purpose of this thesis is to find out how to develop a competitive Tower Defense game on this expanding market.

A Tower Defense game is not a new concept. There are several games of this type with different themes and gameplay already on the market. In order to provide a substantial foundation to start develop for developing a new version of a Tower Defense game from, many of the existing games were examined thoroughly to determine which features to keep and which to avoid. In addition to this, interviews were conducted, with people that play Tower Defense frequently, to find out if they lacked any particular functionality when they played Tower Defense games.

The conclusion of the research was that in order to create a unique gameplay, all of the features of the device would have to be used to some extent. Smartphones primarily provides two features that was taken into consideration when developing the Eskimo Tower Defense; the touchscreen and the accelerometer. The touchscreen provides a new interaction method in comparison with similar games on a computer platform. This is however not a new feature on the smartphone platform. To distinguish the Eskimo Tower Defense game from similar games on the same platform, the mobile phone's accelerometer was used.

Contents

Vocabulary

Mob type: Each mob type has some unique characteristics. In the current implementation of the game there are four mob types, but more types can be added easily when the game is extended. Some mob types have special abilities; others are only characterized by the values of their standard attributes, such as health or armor. Visually a mob type is recognized by a unique look.

Mob wave (or wave): A mob wave consists of one or more mobs that will enter the game as a group. All the mobs in the wave are of the same type. The mobs enter the game one by one and walk the path in a line. There is a few seconds between each wave.

Tower: Towers are built to prevent mobs from reaching the end of the road. A tower can have different types; normal, splash, slow and air. They all have different graphical represenation like brown Eskimo, purple Eskimo, snowman and igloo.

Tower level: A tower have several upgradable levels. For each level it is getting stronger.

Track: A track has a background image, a mob path, a set of mob waves and a current high score. The game has several tracks, which are placed along the progression route on the progression map. To complete a track the player must survive through all mob waves. Each time a mob manages to reach the end of the path without being killed, the player loses one life. If the player's lives reaches zero, the track is lost and the player has to replay it.

Track progression map (or progression map): The progress map visualizes the player's progress in the game and lets him choose which track he wants to play. The progress map resembles a geographical map where each track is a geographical site placed along a route. The progress map is customized to match the theme of the game. In our implementation it depicts a route from the North Pole to The Sun.

Progression route: The progression route is a fixed route that dictates in what order the tracks must be completed by the player. The route is visualized on the track progression map. Each track in the game is placed somewhere along the route. To be able to play a certain track the player must first complete any tracks before it. The route may be forked, in which case only one way leading to a track needs to be cleared in order to play that track. The playerś current progression is visualized with icons.

Player: The player is the physical person who is playing the game.

OpenGL (Open Graphics Library): Is a programming interface to write applications with computer graphics. Often used to write 3 dimensional games.

1. Introduction

1.1. Background

The last year saw a powerful development of so called Smart-Phones in the western world. The introduction of iPhone by Apple Inc. was the starting signal of this rapid development. Since then, several other developers have released this type of phone. These smartphones (välj en stavning!) have operating systems from Apple, Windows or Google. What they all have in common is that it is relatively simple to develop software for them and anyone with basic programming skills can introduce their software onto the phone.

Coupled with this is an expansive system for marketing these applications, where users themselves can make their work available to others. The applications can be sold for money or given away for free. These new phones also make extensive use of alternative input methods such as touch-sensitive screens, expanded voice commands and an accelerometer which detects which way the phone is moved or tilted. [**?**]

A Tower Defense is a common type of game often found as Flash-applications on websites. They all differ somewhat, but what they all have in common is that they feature creatures, organized into waves that try to go from point A to point B. The objective of the game is to prevent this from happening, which is achieved by constructing towers that fire automatically and intermittently upon creatures that enter their range. The game is often varied by introducing different paths these creatures can take, some games even allowing the players themselves to construct obstacles that force the creatures to take a certain path which allows your towers additional time to fire. The game often features different towers with different purposes, as well as different creatures with varying attributes, weaknesses and strengths.

The development of a game was chosen because it represents a lot of the common obstacles encountered when developing software. It provides you with the challenges of drawing on a canvas, managing the efficiency of the code to not slow down the device as well as implementing a good system for user interaction. A game is also a good place to start out when experimenting with new input methods. The concept of tower defense was chosen for the fact that there were few if any tower defense games that featured the use of an accelerometer in its game design.

1.2. Purpose and delimitations

The purpose of this project is to investigate how the new interaction possibilities of smart-phones could be used for a real-time game environment. The goal is to develop the basis for a commersially viable game for the android platform that makes use of the touch screen and accelerometer in new and innovative ways.

The main focus of the project is to develop a stable, extendible and correctly implemented structure for the game, so that further development is facilitated. The number of tracks, tower types, mob types and different sound effects is intentionally small, since focus lies on functionality rather than quantity.

Phone model compability has not been a main focus. Since the Android platform is designed to provide developers and phone manufacturers with an good foundation on which to base their software, the platform inherenltly has a good degree of cross-model robustness. However, in order to ensure proper compatibility, we tested the software on three different phone models (HTC Hero, HTC Legendand HTC Desire), and made some necessary adjustments to accomodate to different screen sizes. Another difference between models is the CPU power, and this fact was not taken into consideration during the project. More information on this can be found under "future work".

2. Theory

2.1. Android architecture

Developing for the Android platform is done using the programming language Java. All calculations are written exactly as done when developing normal desktop applications. However, the Android operating system has very large influence on how applications are executed on the system. Most operating systems on PCs normally allow the user to run several applications concurrently in different windows that also can be viewed simultaneously. On an Android device, there is no native way of seeing what applications are running. The hardware buttons on the device are used to either close applications or send them to the background. Since there is no feedback on what happened to the application, it is important to handle such events in a consistent manner.

Gaining access to the surface of an Android device requires an implementation of the activity class. The first describing line in the Android API about activities is "An activity is a single, focused thing that the user can do CITE HERE. Basically, if you want to create an application that has to show something to the user you have to implement an activity. If you do not need to display anything you may use the service class, used for applications that can run in the background.

As can be seen in figure 2.1, onCreate(), onStart() and onResume() are all invoked as an activity is first created. Several other methods are also invoked whenever the operating system needs to manage memory shortage. Once the activity is up and running, it is important to handle these methods correctly. For instance, when a lot of variables are instantiated in the onStart()-method, memory leaks might occur if they are not set to null in the onStop()-method. Memory leaks can cause the entire device to slow down, which is something that can be very frustrating for users.

The graphical layout of an activity consists of classes called views. Views can be defined either procedurally while creating the activity, or by accessing predefined layouts from xml-files. In Android applications, views are both responsible for drawing images to the screen and for taking care of events generated from user interaction. For instance, a button is a view that can register listeners for onClick-events. Similarly, events generated by the trackball, hardware buttons and touch screen are also handled by views.

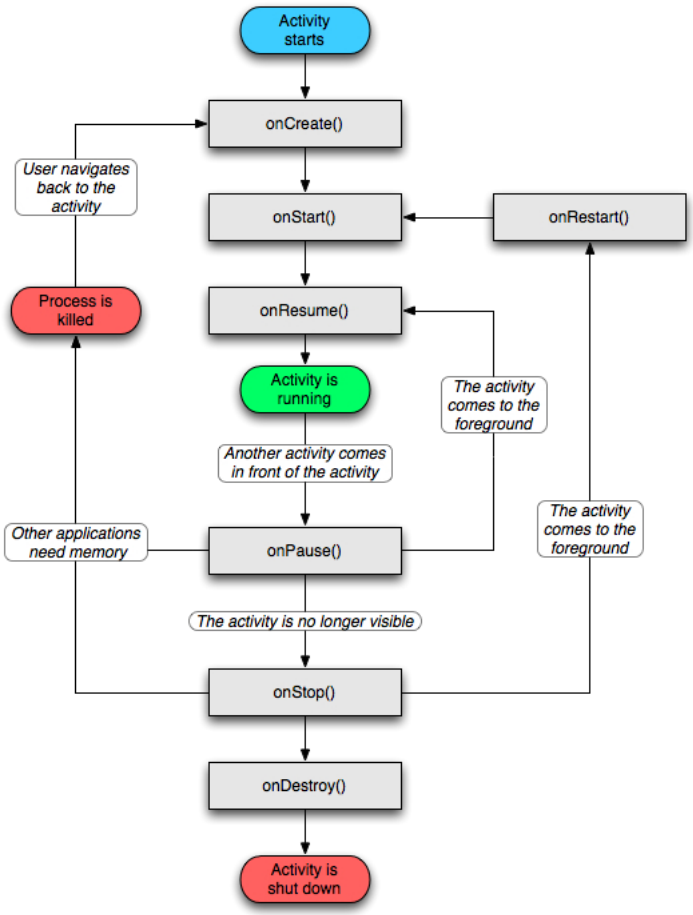

Figure 2.1.: Life cycle of an Activity

2.1.1. Resources

External resources are often used when developing for Android. Any type of file can be included to the binary files when building an application. If Eclipse is being used to develop the application, a file called R.java is generated whenever the resource directory is updated. This file contains translations from integer resource pointers to variable names that are easier to understand. When the application is built, the resource files are compiled into binary files that load fast and efficiently.
(http://developer.android.com/guide/topics/resources/resources-i18n.html)

The Android API (Android, 2010) specifies what directories are allowed in the root resources directory:

- /res/anim/ This directory may contain XML files that describe the behaviour of animations.

- /res/drawable/ This directory may contain image files of the formats png, jpg or gif.

- /res/layout/ This directory may contain XML files describing screen layouts.

- /res/values/ This directory may contain XML files used to define values. As an example, text strings that are used in layouts may be defined here.

- /res/raw/ This directory may contain raw files. Files like these may contain relevant data that isn't suitable to represent in XML files.

Accessing resources is done by invoking the method getResources() on the Context that is attached to the application. Context is an interface that is implemented by fundamental Android classes, such as Activity or Service. As stated in the Android API (Android 2010), " It allows access to application-specific resources and classes".

2.1.2. Data storage

The file system on Android differs from systems used on personal computers. On a computer, file system data files for one application can be read by any other application. On Android however data files created by one application is only readable to that application. Android has four different solutions to store and receive data from the file system on a mobile phone; preferences, files, databases and network. """"(Android 2010)""""

The preferences solution uses key-value pairs to write simple data types, such as texts to be loaded at the start of an application, or settings the user wants to save for next time he or she starts the application. This data is a lightweight method of writing and retrieving data, and is therefor recommended to use for simple data types. """"(Android 2010)""""

Another way to manage data storage is to use files. This is a basic way to handle data, where files are created and written to, and read from, the mobile phone's memory card. """"(Android 2010)"""""

Android also comes with the possibility of using databases for data storage. The type of database available on Android is SQLite. (Android 2010) SQLite is a lightweight database engine, built to suit devices with limited memory. It read and writes to files on the device's file system. "A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file." """""(SQLite 2010)"""""

As long as the phone is connected, either to a 3G or a WiFi network, it is possible to use the network connection to send and receive data. """"(Android 2010)"""""

2.1.3. XML

XML which is an abbreviation for eXtensible Markup Language, is used for representing arbitrary data structures. In many areas XML is used to just store and retrieve data like a database, even if it is not as powerful as an SQL (Structured Query Language) database. For example the waves and paths for the game are stored in XML files.

When working with the Android platform, storing and retrieving data is not the only thing you can do with XML. There is a built in XML-editor in the SDK (Software Development Kit) (sätt i vocabulary istället?which you can use to create Android activities. An activity is the window that will be shown on the screen. Inside that window is where all the graphics are put. The editor is very easy to use as it uses the drag-and-drop concept. You have several layout options like GridView, ListView, LinearLayout etc. to choose between and combine. There are also several view options like normal View, Button, Checkbox, TextView and much more. As mentioned before, items are dragged and dropped to the correct positions. There is also a property window for every item that gives access to customizing that particular item in the layout.

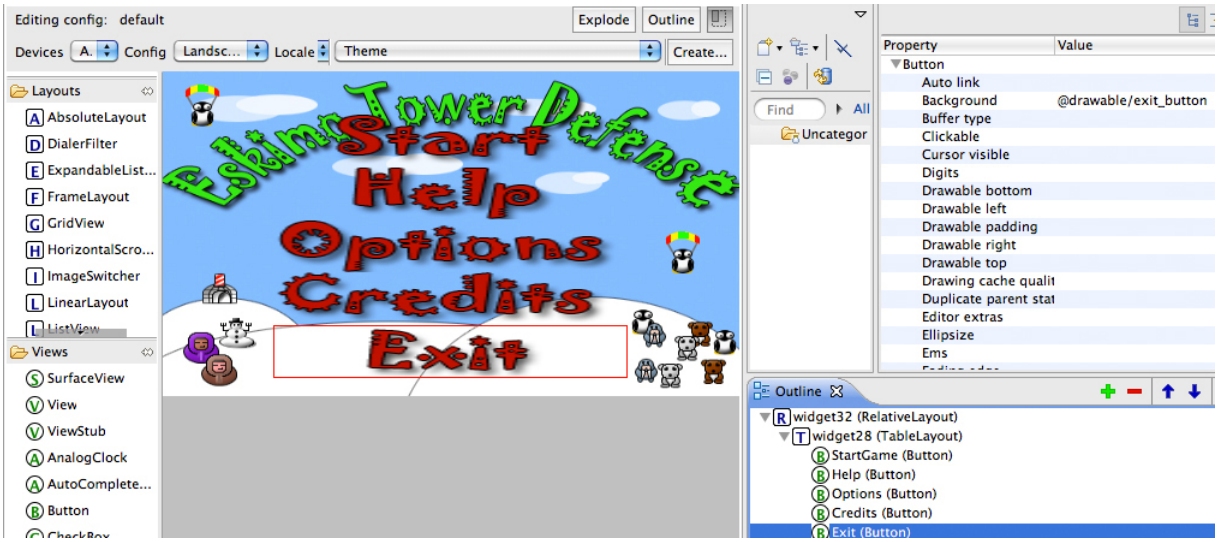

Figure 2.2.: Caption for the xmleditor

For each activity using a predefined layout, there has to be an XML file describing the layout. You can either create the whole layout in the XML-editor or just put a Relative Layout and solve the graphical issue with other techniques.

2.1.4. Graphics

There are three ways to handle graphics when working with Android. The first one is as mentioned above, using predefined layouts in XML files. The other two are with

Canvas or OpenGL. Canvas provides simple tools like canvases, rectangles, color filters and bitmaps which let you draw pictures on the screen. It is handled like layers even if it is not exactly layers. The last object that is drawn on the canvas will be shown at the top. You only have to state where on the screen it will be drawn. Canvas only supports two axis, x and y coordinates versus OpenGL which has full support for 3D programming. That should be the main reason when choosing between them.

2.1.5. Sound

3. Method

Text...

3.1. Interviews

Text...

3.2. Agile development

Text...

3.3. Git - A version control system

Text...

3.3.1. Git versus SVN

The decision to use Git instead of SVN was partly based on the fact that the group had greater experience using Git. Also, since Git allows the project to be stored locally on every members computer, work can be done on the implementation despite lack of internet access. This also means that any system failure is only going to affect one individual, who could then fetch an updated version of the project from the other members, providing the project with an increased tolerance to any system breakdowns. Other common reasons to use Git is that it uses less space, and completes its tasks faster, since it is quicker. While SVN's graphical user interfaces lends itself to a more speedy learningprocess, the amount of time needed to learn and utilize Git was not notable.

3.4. Code convention

Text...

3.4.1. Style conventions

Text...

Javadoc and comments

Short methods

Long methods were broken down into shorter ones to increase readability, where possible. One good example of this is the method onDraw in the class GameView, which calls several sub-methods that draws different parts of the game, instead of drawing everything inside one single method.

```
public void onDraw(Canvas canvas) {

        drawBackground(canvas);
        drawSplashWater(canvas);
        drawTowers(canvas);
        drawMobs(canvas);
        drawSnowballs(canvas);
        drawProjectiles(canvas);
        drawButtons(canvas);
        drawStatisticsText(canvas);
        drawRewardsAfterDeadMob(canvas);
        //----more code-----//
}
```

Figure 3.1.: Caption for onDraw...

3.4.2. Programming practices

Text...

3.4.3. Android best practices

Text...

3.5. Android development environment

Text...

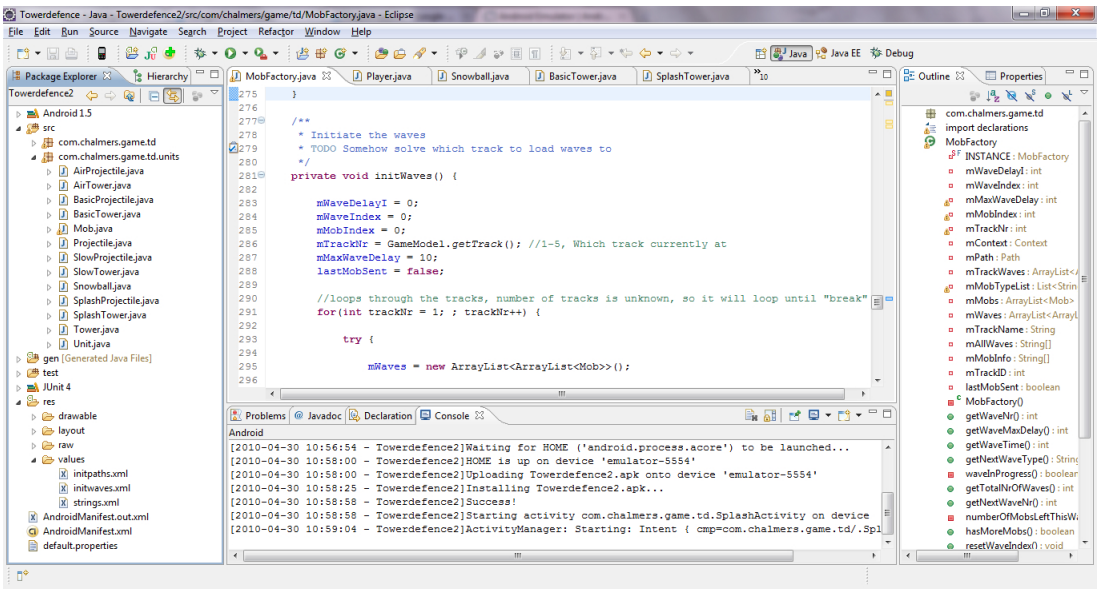

Figure 3.2.: The eclipse Intergrated Development Environment

3.5.1. Eclipse and Android SDK

Text...

Figure 3.3.: The Android emulator

4. Results

4.1. The resulting game: Eskimo Tower Defense

Text...

4.1.1. Structure

Text...

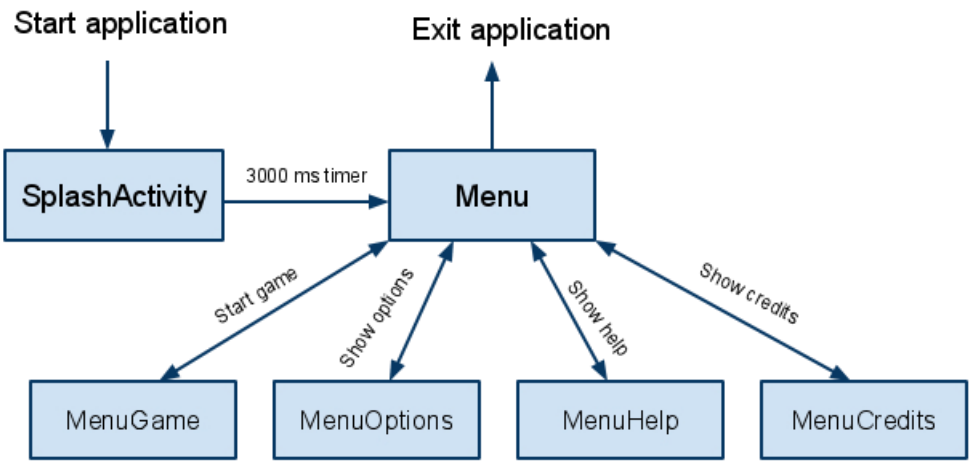

Figure 4.1.: Flowchart of activities in Eskimo Tower Defense

```
while (mRunThread) {
    canvas = null;
    try {
        canvas = mGamePanel.getHolder().lockCanvas(null);
        synchronized (mGamePanel.getHolder()) {
            mGamePanel.updateModel();
            mGamePanel.updateSounds();
            mGamePanel.onDraw(canvas);
        }
    } catch (InterruptedException ie) {
    // doNothing();
    } finally {
        // do this in a finally so that if an exception is thrown
        // during the above, we don't leave the Surface in an
        // inconsistent state
        if (c != null) {
            mGamePanel.getHolder().unlockCanvasAndPost(canvas);
        }
    }
}
```

Figure 4.2.: Caption for GameThread

UML

Text...

4.1.2. XML layout

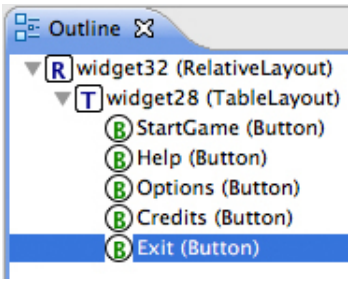

Figure 4.3.: Outline view of the layout editor in eclipse

Text...

Property	Value
Button	
Auto link	
Background	@drawable/exit_button
Buffer type	
Clickable	

Figure 4.4.: Property window of the layout editor in Eclipse.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item    android:state_focused="true"
             android:state_pressed="false"
             android:drawable="@drawable/exit_highlighted"/>
    <item    ...
// Other images for different states of the exit button...
</selector>
```

Figure 4.5.: Caption for exit button

4.1.3. Graphics

Text...

Draw

Text...

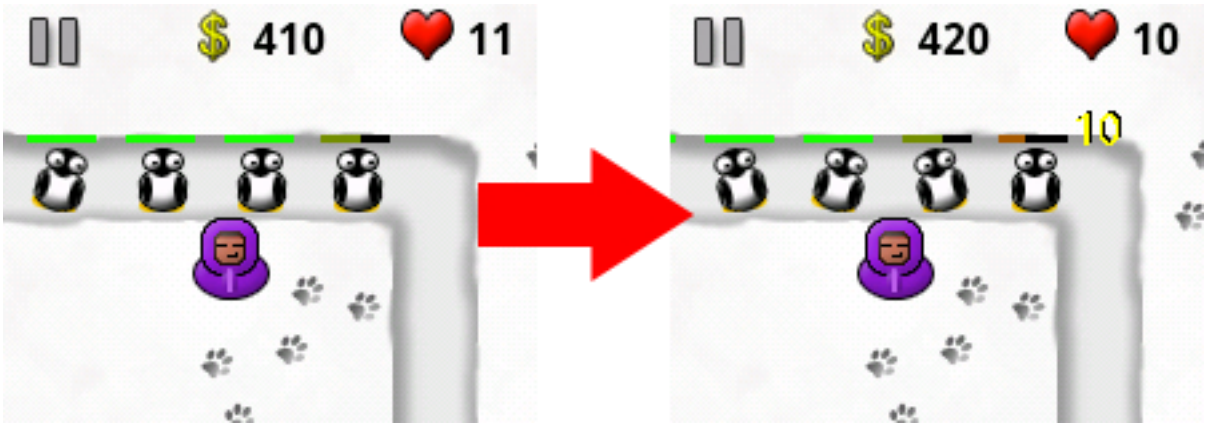

Figure 4.6.: Screenshot of a mob dying.

```
private void drawRewardsAfterDeadMob() {
    for (every reward) {

        draw the reward graphic
        change y-coordinate of reward
        increase reward frame variable

        if the reward frame variable is higher than 12
            remove the reward
    }
}
```

Figure 4.7.: Example of how to draw graphic to the screen

Animation

Text...

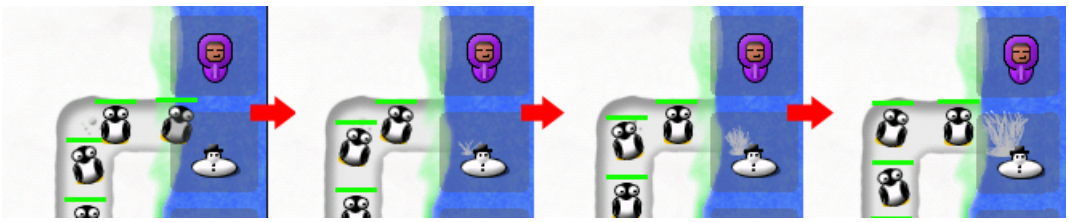

Figure 4.8.: Screenshot of water splash animation

```
private void drawSplashWater() {
    if(water animation step is between 0 and 5) {
        draw first image of the animation
    } else if(water animation step is between 5 and 10) {
        draw second step of the animation
    } else if(water animation step is between 10 and 15) {
        draw third step of the animation
    } else if(water animation step is between 15 and 20) {
        draw fourth step of the animation
    } else if(water animation step is between 20 and 25) {
        draw final step of the animation
    }
    increase animation step
    if the final animation step is reached, remove the animation
}
```

Figure 4.9.: Caption for draw water splash..

Menus

Text...

Tower placement

Text...

Tower upgrade

Text...

Mobs

Text...

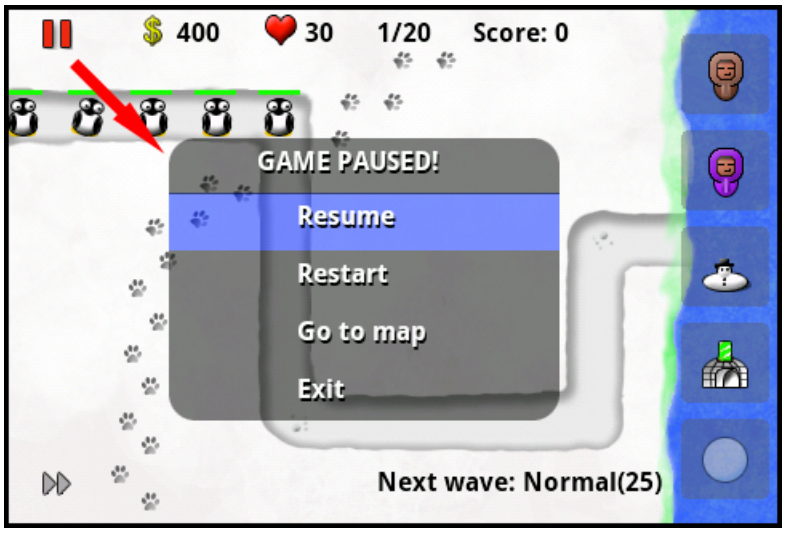

Figure 4.10.: Screenshot of in-game pause menu.

```
if (pause button is pressed) {
    GAME_STATE = STATE_PAUSED;
}
```

Figure 4.11.: Caption for in-game pause menu codesnippet...

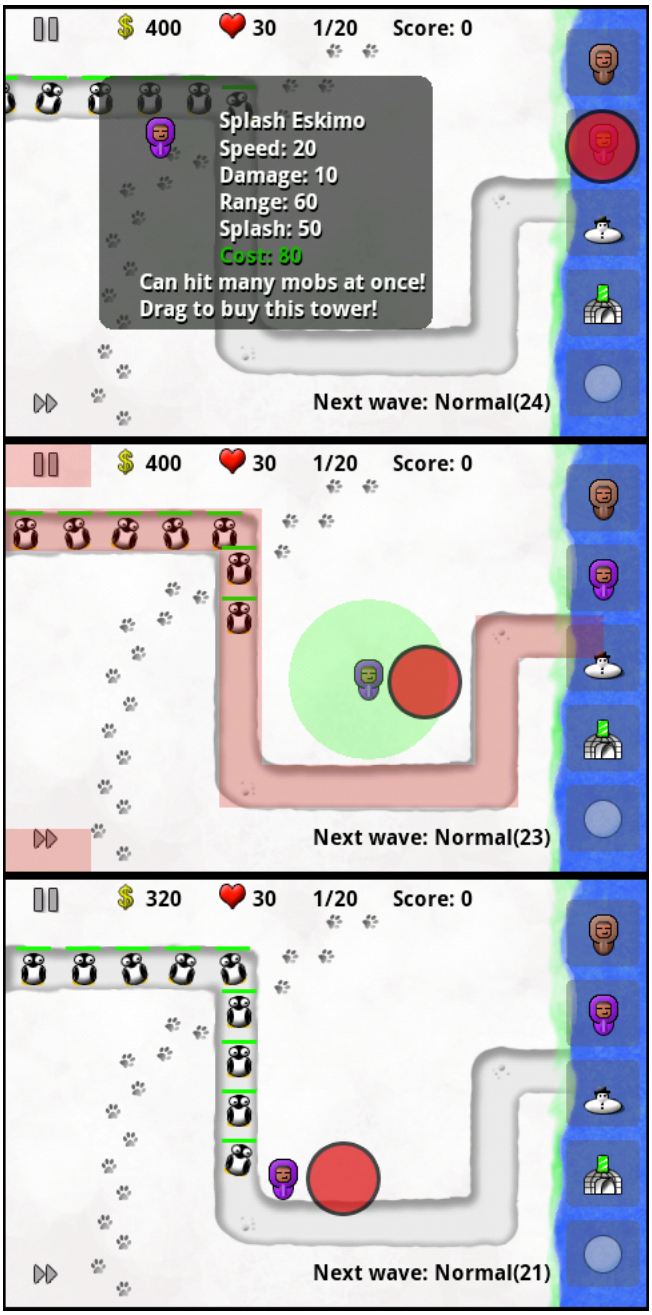

Figure 4.12.: Screenshots of a tower being built.

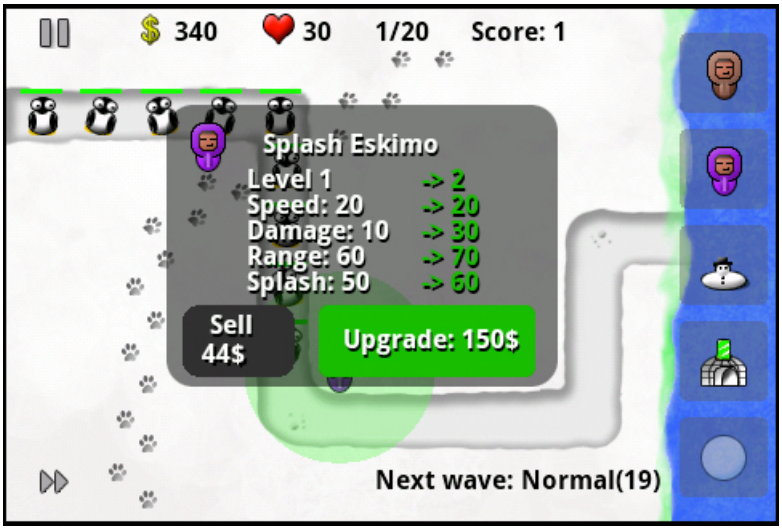

Figure 4.13.: Screenshot of upgrade window

Figure 4.14.: Caption normal tower

Figure 4.15.: Caption

Figure 4.16.: Caption

Figure 4.17.: Caption

Figure 4.18.: Caption

```
int hpRatio = (int)(255* (double)m.getHealth() / (double)m.getMaxHealth());
```

Figure 4.19.: Caption

4.1.4. Progression map

Text...

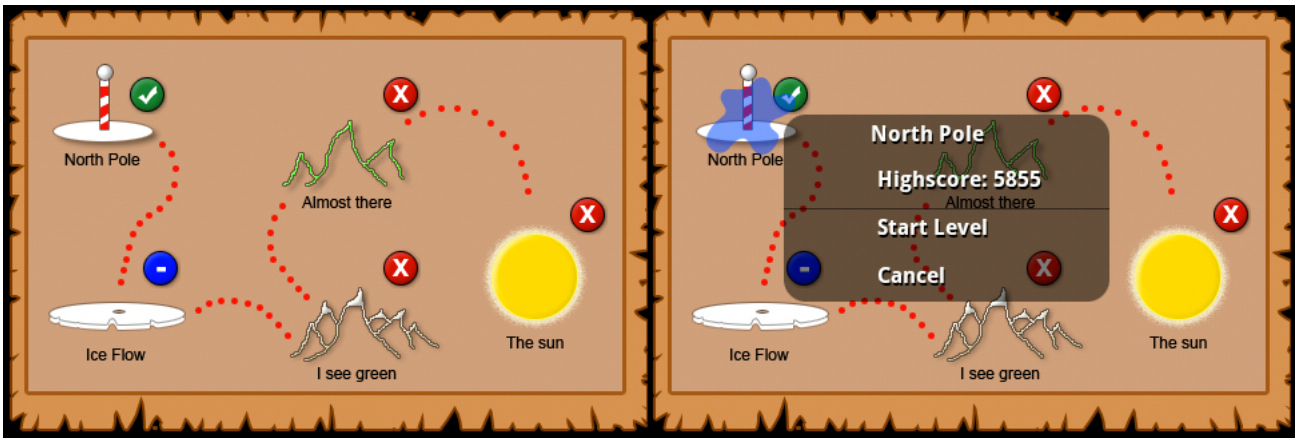

Figure 4.20.: Caption

4.1.5. Tracks

Text...

4.1.6. Physical buttons

Text...

Figure 4.21.: Hardware buttons on a HTC Hero

```
public boolean onKeyDown(int keyCode, KeyEvent event) {

        switch (keyCode) {
            case KeyEvent.KEYCODE_MENU:
                // Handle hardware menu button
                GAME_STATE = STATE_PAUSED;
            break;
            case KeyEvent.KEYCODE_BACK:
                // Handle hardware "back" button
                GAME_STATE = STATE_PAUSED;
            break;
            case KeyEvent.KEYCODE_VOLUME_UP:
                // Increase the volume of the game music
            break;
            case KeyEvent.KEYCODE_VOLUME_DOWN:
                // Decrease the volume of the game music
            break;
        }

        return true;
}
```

Figure 4.22.: Caption

4.1.7. Game play

Text...

```
if (back to menu button is pressed) {
    // go back to main menu
    // stop current game thread
    // release references
    // shut down current Activity
}
```

Figure 4.23.: Caption

4.1.8. Units

Text...

```
public static double getDistance(Coordinate c1, Coordinate c2) {
    double tx = c1.getX();
    double ty = c1.getY();
    double mx = c2.getX();
    double my = c2.getY();
    return Math.sqrt((tx - mx)*(tx - mx) + (ty - my)*(ty - my));
}
```

Figure 4.24.: Caption

4.1.9. Towers

Text...

Subclasses

Text...

4.1.10. Mobs

Text...

4.1.11. Waves

Text...

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="mobs_track_1">
        <item>[MobType] [Amount of mobs] [MobHealth]</item>
        ...
    </array>
</resources>
```

Figure 4.25.: Caption

Text...

```
for(int trackNr = 1; ; trackNr++)
```

Figure 4.26.: Caption

Text...

After the string has been processed, a third loop creates the given amount of mobs and stores them in an ArrayList. When all mobs are created the list of mobs is stored in another ArrayList to keep track of every tracks different waves.

```
mMobInfo = mAllWaves[waveIndex].split(" ");
String sType = mMobInfo[0];

if(sType.equals("NORMAL"))
     iType = Mob.NORMAL;
else if (sType.equals("AIR"))
    iType = Mob.AIR;
else if (sType.equals("FAST"))
    iType = Mob.FAST;
else if (sType.equals("HEALTHY"))
    iType = Mob.HEALTHY;
else // if (sType.equals("IMMUNE"))
    iType = Mob.IMMUNE;

int numberOfMobsInWave = Integer.parseInt(mMobInfo[1]);
int health = Integer.parseInt(mMobInfo[2]);
```

Figure 4.27.: Caption

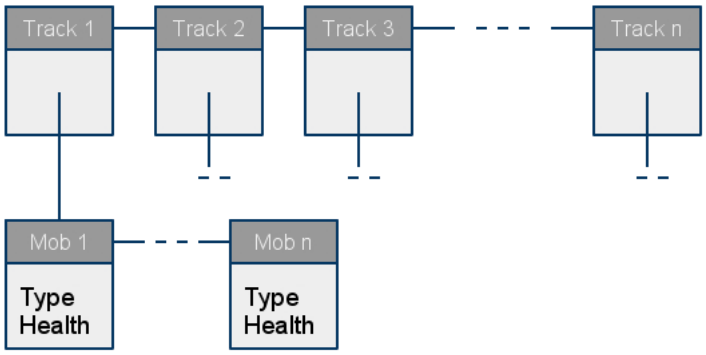

Figure 4.28.: Data structure of a wave object

```
if (!lastMobSent && mWaveDelayI >= mMaxWaveDelay)
    return mTrackWaves.get(mWaveIndex).get(mMobIndex);
else
    mWaveDelayI++;
    return null;
```

Figure 4.29.: Caption

4.1.12. Path

Text...

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="path_track_1">
        <item>[X coordinate] [Y coordinate]</item>
        ...
    </array>
</resources>
```

Figure 4.30.: Caption

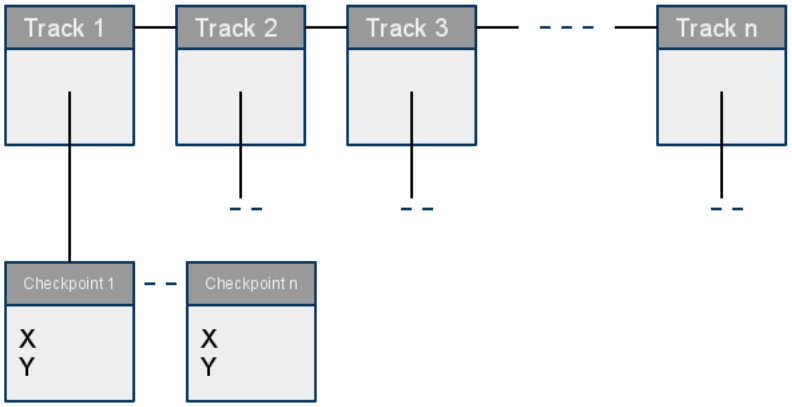

Figure 4.31.: Caption

4.1.13. Snowball - Accelerometer based interaction

Text...

```
// Access the sensor service of the system
if (the list of sensors contains an accelerometer sensor)
    // Register a listener to the accelerometer sensor
```

Figure 4.32.: Caption

Text...

```
private SensorEventListener listener = new SensorEventListener() {
    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
    public void onSensorChanged(SensorEvent event) {
        mLatestSensorEvent = event;
    }
};
```

Figure 4.33.: Caption

Text...

```
public void updatePosition(SensorEvent s) {

    setSpeedX(getSpeedX() + s.values[1] / 45);
    setSpeedY(getSpeedY() + s.values[0] / 45);

    setX(getX() + getSpeedX());
    setY(getY() + getSpeedY());
}
```

Figure 4.34.: Caption

4.1.14. Projectiles

Text...

Subclasses

Text...

4.1.15. Money and high-score

Text...

```
if (Mob has walked less than 500 pixels) {
    // The new score is added to the existing score
    // Using the algorithm:
    // Old score + (Mob max health / 10) *
    //      (the distance the mob has walked / 500 pixels)
} else {
    // Old score + (Mob max health / 10) * 0.5
}
```

Figure 4.35.: Caption

Text...

```
try {
    // read from file tddata.txt
} catch(FileNotFoundException fnfe) {
    // create file tddata.txt
}
```

Figure 4.36.: Caption

5. Conclusions

Text..

6. Discussion

6.1. Concept

6.1.1. Maze versus path

The first design problem we faced was how to make the path. We could either make a game with a predetermined path or having the mobs choose their own closest path to the finish. We had during the testing of different games tested Robo Defence(Lupid Labs for Android. This a TD that had no static path, but instead you create the path when you build your towers. This allows you to experiment with different paths to find the best one, like a maze. If we use the static path the user cannot control which way the mobs will go.

The benefits with a static path is that the game is easier to play, you do not have to think about how the mobs will move based on how you build your towers. The game will look more attractive with a path drawn on the map from the start. If you build your own path you have to have open spaces and the map might look empty and boring. Another drawback with a maze is that it is more complex and might scare new players away.

The benefits with a maze is that it is more challenging. The game play is often different from player to player because the may build different mazes. The game also gets another dimension when you place cheep tower not to damage the mobs but to increase the length of the path. The flying mobs also complicates the strategy when they can fly over the towers and then not the path created by the player.

When discussing and voting we decided to go for the static path design. Since we have never developed a game before we did not want to code a too complicated A.I. The static path design seemed to be the easiest to make and if we wanted to increase the complexity we had several ideas how to make the game more unique.

6.1.2. Theme

One of the bigger discussion subject in the group was what kind of theme the game should have. Since no one in our group was really good at graphics we waited a long time with deciding the theme. At first we only had example pictures for testing the

rest of the functionality of the game. We later changed this to a winter theme just for having a temporary theme that did not look too bad. Since we had not theme at the start we named our sub classes of towers and mobs based on their function, not their apperance.

We agreed that the game should have a background story that the theme should be linked to. In the first real discussion we decided to go for a green house effect-theme. The base story would be that the animals of the north pole would flee from their melting environment. The map would then have different themes; like poluting factories, melting iceblocks and smog.

We later changed this theme to a Eskimo theme. The background story was the same but instead of the green house effect the animals was migrating south just because they wanted to. The eskimos does not like this and therefore tries to stop them. The maps will then gradually become greener and greener when progressing thought the game. The towers should be different eskimos, a snowman and an igloo canon. The mobs is different animals like pingvins, ice bears and wallruses.

We also had a special ball that could be rolled over the map. This would be drawn out like a big snowball, rolling over the animals.

6.1.3. Unique features

We wanted to make our game stand out from other Tower Defences to make it more desirable. Modern Android phones offer many new ways to controll your phone which we wanted to take advantage of. The idea we had was that we wanted to add an extra dimension to the game using the phones accelerometer. One of the ideas was to control the towers by tilting the phone. Another idea was that the player should be able to controll the range of the towers. If the player tilts the phone to the left, the towers range will increase in that direction. Another idea was to control the speed of the mobs. When tilting to the left, it would create a slop to the left making the mobs moving left go faster and mobs moving right slower.

We also had an idea to make the path slips into to paths. We have not seen any other games that has multiple paths for the mobs so this would be an easy feature to implement that would make it more unique. We also thought of letting the user control with way the mobs go in a crossroad using the accelerometer. The more you tilt in one direction, the higher is the probability that the mobs go that way.

The idea that we actually implemented was that of a snowball. The snowball will roll over the map controlled by tilting the phone, almost like the classical board game Labyrinth. In It is a bonus that will be given to the player from time to time while playing. The player can use this snowball to kill mobs when in difficult situations. We had some discussion how the snowball would interact with the units on the map. Firstly we had it kill every mob it touched. This was the easiest was to code it but it made

the snowball too good, especially against bosses who had much more health than other mobs. We later changed this so it would take a percentage of the mobs health every frame it touched it. We also set the snowball to take a less percentage on bosses to make it more balanced compared to normal mobs. The downside of this approach is that the snowball takes less and less damage of one mob while currently on it. This is because it takes a percentage of the mobs current damage. When the mob has high health the snowball inflicts high damage and when it's health is low it inflicts low damage. This makes the snowball more effective against healthy mobs which can be quite confusing for the player.

We also had a discussion if the snowball should have any negative aspect to make it harder to use. One idea was to have the snowball not only damage the mobs but also the towers. This combined with a more powerful snowball would make it both effective in hard situations but also a risk. The problem with this is that the player then could place the snowball on the path and without tilting the phone killing the mobs very effective. This would discourage the user from tilting the phone and use the snowball as it was intended. This is also a problem even without the snowball killing the towers but then the snowball would be balanced to take less damage.

6.1.4. Waves

A big issue was how the waves was supposed to work. In other TD games like Bloons and Element TD the game pauses between every wave, giving time for the user to build more towers. Another way is to make the waves come continuously but with a delay between them. We decided to go for the delay approach. This to make the game flow better and to not have the player press a lot of times. Since we wanted to have a lot of waves this might be irritating for the player to always have to press a button before each wave.

The negative aspect with this approach is that the mobs have different speed. This may result in that one wave catches up with an earlier wave. This is very noticeable for the boss waves. A boss wave consists of on mob with much more health than a normal mob.This results in that it takes more time to kill. Since there only is one mob on boss waves the countdown to the next wave start immediately after it is created. To solve this we simply increased the delay after slow boss waves.

6.1.5. User interface

User interface is one of the most important areas when working with touchscreen mobile phones or small touchscreens in general. You want to include as much as possible on the screen but at the same time you can not make the items too small. Since you use your fingers to interact with the phone and some people might have bigger fingers then others. You have to implement enough big buttons to satisfy all users.

The way you play the game is in landscape mode, landscape mode is when you hold your phone in a 90 degree angle. The way you hold it, only leaves your thumbs to interact with. That means you are forced to make big buttons.

6.1.6. Graphics

Canvas vs Open GL Even if you can do 2D games with OpenGL too. The decision was made to to work with Canvas since there was to be no implementation of any 3D graphics in the game. Some of us was also familiar with the Canvas concept which even made it an easier choice.

Animations

Interface

6.1.7. Data storage

6.2. Game balance

One of our goals was to make a game that was fun game that would appeal to a customers market. Through our interviews We found that an attractive game should both be challenging and easy to learn. The interviewees also said that here would be good if one had to use different type of strategies for different situations. The should not one strategy or one tower that was best in all situations. To do this We had to balance the game so the towers had different abilities. The game difficulty should also increase in a good way. The first map should introduce the player to the game and not scare him away. The game should also provide a challenge and the player should not be able to finish it to easily. Our balancing mainly consisted of changing the following variables in the game.

6.2.1. Towers

The balancing of the towers was done to make sure that no tower was superior to another. If so the game would be boring if you could finish the game by only building on type of tower. Our idea was that the player needed to build different towers for different situations. This is one of the reasons for having different types of towers and mobs. Different maps contains different combinations of mob types and there for the user has to use an appropriate strategy.

6.2.2. Mob waves

The biggest part was the balancing of the mob waves. To make the game fun to play We had to test with some values, play through the game and see how easy it was. Each killed mob gives a specific amount of money. To make the game balanced the income of money should be similar to the cost of building towers to defend yourself. If you get to much money the game would not be challenging and if you get to little money it would be impossible to finish.

Since the maps should have different difficulty We had to manually set the health of the waves for each map. The waves of the first map should be easier than the waves of the second map and so on.

6.2.3. Snowball

The player get one snowball for each 4000 points that is collected. The reason for the amount of points (4000) is that you normally get around 10000-15000 points on a map. We did not want the user to get the snowball to often an 2-3 times per map sounded good.

The damage of the snowball was also up for discussion. We first had the snowball kill the mobs instantly but later changed this so it killed 8% and the mobs current health each frame. This resulted in a snowball that was good balanced. The snowball was also modified to take a less percent for each frame for the bosses. This because the bosses have around 15 times more health than normal mobs. If the snowball would take the standard percentage from the bosses the snowball would be unnaturally good for bosses.

7. Future work

Some additions to the game were left out of the project scope, but should be added to increase the value of the finished game. The game that resulted from this project is playable in its current state, and it is, as stated before, an excellent basis to build a commercial game from. For those who would like to do that, there are some things that should be taken into consideration, because they were intentionally left out. These are fixed time steps, blabla, blabla and blabla, and are further explained below.

7.1. Fixed time step

To improve the game and make public high score lists meaningful the game would need to accommodate to different processor capacities, by making the speed of the game independent of the speed of the phone. There are a number of well-known methods available to achieve this effect, but it was excluded from this project in favor of adding features that more directly affect the game experience.

A. Appendix

List of Figures