Vocabulary

OpenGL (Open Graphics Library): Is a programming interface used by software to implement computer graphics. Often used to write three-dimensional games.

XML: eXtensible Markup Language. It is used for representing arbitrary data structures.

SDK: Software Development Kit. A set of software used to enable developing software for a certain platform or purpose.

IDE: Integrated Development Environment. A set of software used by a developer to ease the development process.

UML: Unified Modeling Language. A general language for creating a model for a system. The model consists of the different program classes and the relations between them.

1 Introduction

In the beginning of 2007 a powerful development of smartphones began. Apple's introduction of the iPhone was a starting signal of this rapid development. (Grossman, 2007) Since then, several other companies have released smartphones. The majority of them have operating systems developed by Nokia, RIM, Apple, Microsoft or Google (Canalys, 2010).

Each smartphone operating system has its own system for publishing applications. The applications can be downloaded by users with the same operating system. It is up to the developers to charge a fee for their applications or if they want, distribute them for free. This new way of marketing applications allows everyone from bigger companies to the individual developer to compete under basically the same conditions.

The development of a game was chosen because it consists of a lot of common obstacles encountered during software development. The difficulties of programming a game include drawing graphics, managing code efficiency as well as implementing a good system for user interaction. A game is a good place to start when experimenting with new input methods, because there are no predefined rules for how a game is supposed to work.

Tower Defense is a common type of game, often found as Flash-applications on websites. The general concept is that creatures, organized into waves, are trying to get from point A to point B. The objective of the game is to prevent the creatures from reaching their destination. This is achieved by constructing towers that fire automatically at creatures that enter their range. The games often feature different towers with different purposes, as well as different creatures with varying attributes, weaknesses and strengths.

1.1 Purpose and delimitations

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

The main focus of the project is to develop a stable, extendable and correctly implemented structure, to facilitate further development. The number of tracks, tower types, mob types and different sound effects is intentionally small. Focus lies on functionality rather than quantity.

Game balance has a huge impact on commercial viability. According to the first interview an unbalanced game is perceived as less attractive to play. Due to this fact, one of the goals of this project is to make the game feel as balanced as possible. The game should be easy to play but still provide a challenge for the user.

Phone model compatibility has only been considered to a small extent. The software was only tested on three different phone models (HTC Hero, HTC Legend and HTC Desire). Differences between models in CPU power was not taken into consideration during the project.

2 Theory

Developing for Android is done using the Java programming language, but requires the use of some additional classes and techniques to adapt the program for the operating system. Android also enables several data storage and retrieval options.

2.1 Android architecture

Developing applications for the Android platform is done using standard approaches of the Java programming language.

The Android operating system influences how applications are executed. Most operating systems on personals computers allow the user to run several applications concurrently in different windows, that can be viewed simultaneously. On the Android device, there is no native way to determine which applications are running. The hardware buttons on the device are used to either close applications or send them to the background. Since the user receives no feedback on what happened to the application, it is important to handle such events in a consistent manner, to avoid confusion.

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" (Android, 2010). When compared to applications on a personal computer, the Activity can be seen as a window showing different user interface elements. To create an application that shows something to the user, an Activity must be implemented. If it is not important to display information to the user, the Service class may be used (a class designed for applications running in the background).

As can be seen in figure 2.1; onCreate(), onStart() and onResume() are all invoked when an activity is first created. Several other methods are also invoked when 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, if a lot of variables are instantiated in the onStart()-method memory leaks might occur, unless the variables are set to null in the onStop()-method. Memory leaks can cause the entire device to slow down, which can be very frustrating for the user.

The graphical layout of an activity consists of objects of the View class. Views can be defined either procedurally while creating the activity, or by accessing predefined

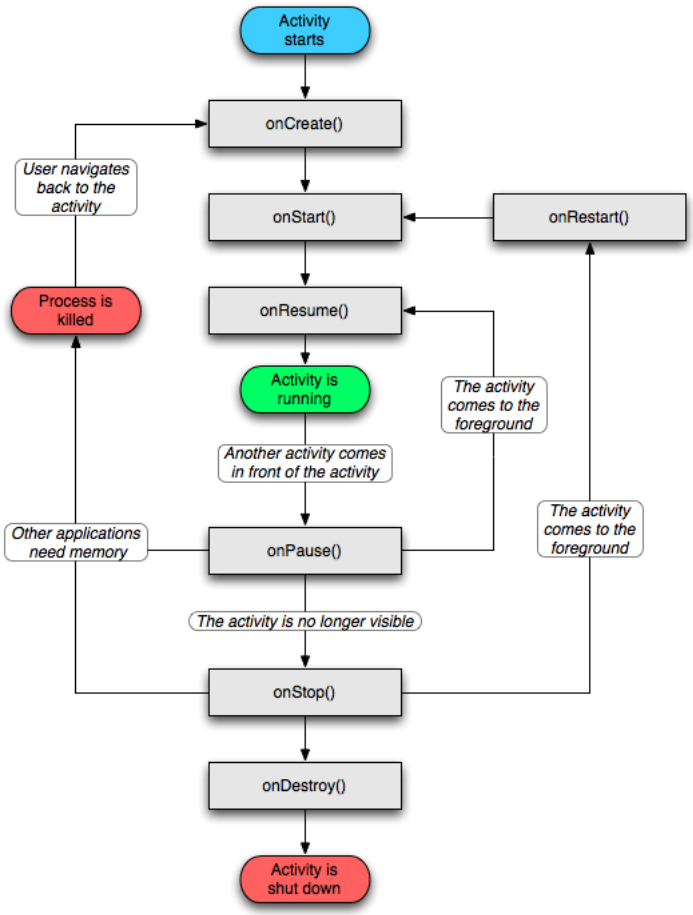

Figure 2.1: Life cycle of an Activity

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 touchscreen are also handled by views.

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 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. (Android, 2010)

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.".

2.1.1 Data storage

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

The preferences solution uses key-value pairs to write simple data types. It can be used for texts that should be loaded at the start of an application, or for settings that the user wants to save until next time he starts the application. This data is a lightweight method of writing and retrieving data, and is therefore 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 on the mobile phone's memory card, where files are created, written to and read from. (Android, 2010)

Android provides the ability to use databases for data storage. The type of database available on a Android device is SQLite. (Android, 2010) SQLite is a lightweight database engine, built to suit devices with limited memory. It reads 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 to the Internet, either via 3G or via a wireless network, it is possible to use the network connection to send and receive data. (Android, 2010)

2.1.2 Graphics

There are three approaches to handling graphics when working with Android. The first approach uses predefined layouts in XML-files. The other two approaches involve the Java class Canvas or the cross-language standard specification OpenGL. The Canvas class provides simple tools like rectangles, color filters and bitmaps that let you draw pictures on the screen. It works in a similar way as layers: the last object that is drawn on the canvas will be drawn on top of anything that was drawn earlier. Canvas only supports two axes, x- and y-coordinates compared to OpenGL that has full support for programming 3D graphics.

For each activity using a predefined layout, there has to be an XML-file describing that layout. There is a built-in XML-editor in the Android Software Development Kit for Eclipse that can be used to create these layouts. The editor shows a preview of the

window that will be shown on the screen of the device (figure 2.2). All the graphics are put inside that window. The editor is very easy to use as it uses the drag-and-drop concept.

There are several layout options, that govern how elements will be placed, such as GridView, ListView and LinearLayout to choose between or combine. There are also several view options such as normal View, Button, Checkbox, TextView and many more. Items are dragged and dropped to their correct positions. There is also a property window for every item, that gives access to customizing that particular item in the layout.

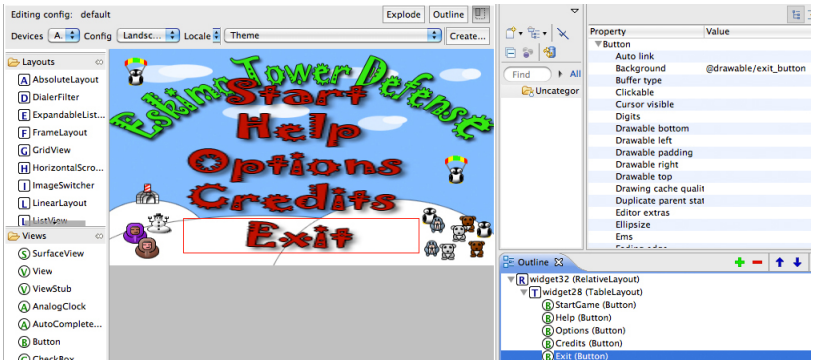

Figure 2.2: The XML editor in Eclipse

2.1.3 Sound

The Android operating system is able to provide playback of different media files. A list of all supported media formats is published in the Android API (Android, 2010). When considering sound files in particular, all of the major sound formats (MP3, MIDI, WAVE, Ogg Vorbis and more) are supported. Unless playing sounds is the main purpose of the application, it is desirable to use files that are small and therefore require less memory.

Sound implementation in Android applications is done by utilizing the classes MediaPlayer and SoundPool. The SoundPool class should be used for small sounds, that are likely to be repeated a lot. Sounds added to a SoundPool are decoded by a MediaPlayer and stored as raw bitstreams. Not having to decode sound files every time they are played reduces the risk of experiencing performance issues due to heavy CPU load. MediaPlayer is better used for long sound files, such as background music. (Android, 2010)

3 Method

Prior to designing the game, background information was collected on the developer team's, as well as other people's, views and expectations on a new Tower Defense game. A number of existing well-known tower defense games were also tested and discussed to get a general idea of which features that are desirable. The design process included discussions of different ideas, drawing of mind maps and draft UML-diagrams.

The development was done in Eclipse IDE with the Android SDK-plugin. The program language Java was used. Included in Android SDK is an emulator which made it possible to test code directly on the computer.

The implementation was done using an agile development approach, in the sense that we worked in small iterations and changed the requirements continuously during the development process. To make the project easier to maintain, the distributed subversion management system Git was used.

Code convention was used to make the code easier to understand within the development group or for otherdevelopers that might work on the system later on.

3.1 Interviews

Interviews were used as a tool in two different stages of the development; the research stage and the final testing stage.

Interviews were conducted to get a better picture of what makes a Tower Defense game good. The interviewees where people that play games regularly and had prior experience of playing Tower Defense games. Ten interviews were conducted, following an unstructured interview template with mainly open questions. The full interview template is found in Appendix XX.

The interviews provided some insight into peoples thoughts and the statements from the participants were used to make important design decisions. The interviews made it obvious that obvious that everyone had their own opinion on which features a good game should include, but several statements were general and helped avoiding common mistakes.

In the final stage of development, ten more interviews were conducted to evaluate the resulting game. The template can be found in Appendix XX. The answers helped finding areas that needed improvement.

3.2 Android development environment

Developing Android applications requires Java Development Kit (JDK) and an Integrated Development Environment (IDE). Google recommends Eclipse with Android Development Tools (ADT). The ADT is a plugin for Eclipse that allows easy creation and management of Android projects. (Android, 2010)

The Eclipse Foundation is an open-source community originally created by IBM in 2001. Its most popular IDE Package is the Eclipse IDE for Java Enterprise Edition (EE) Developers that currently have over 1.2 million downloads. Eclipse IDE is free to use and allows the users to easily create Java applications. (**?**)

Figure 3.1 shows a screenshot of a project in Eclipse. The left panel displays the project files and the center panel displays the currently open file. At the bottom there is a console showing the status of the different tasks that Eclipse is performing: log messages, error messages etcetera. To the right is an outline of the methods and variables in the current file. This helps the user to easily overview and navigate the code.

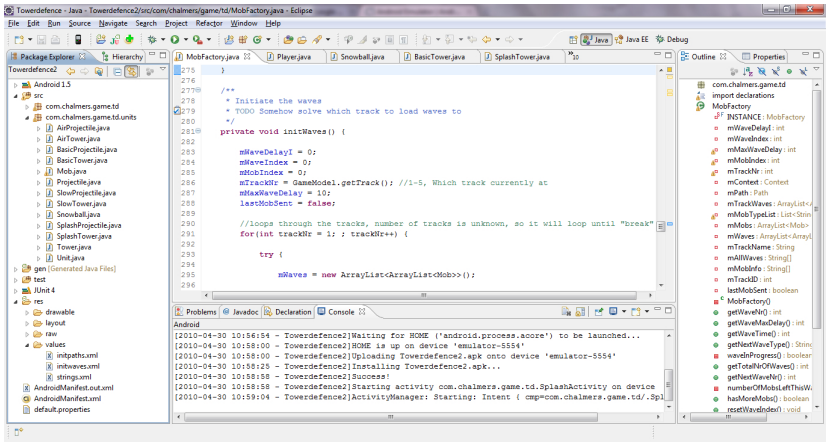

Figure 3.1: Eclipse Intergrated Development Environment

The Android Development Kit includes an Android emulator (figure 3.2). This allows the developer to test her applications without the use of an actual phone. It supports a majority of the functionalities of a real phone, such as simulating SMS, phone calls, events and geographic locations (like GPS navigation). The mouse can be used to click on the screen to emulate usage of the touchscreen of the device, and virtual buttons represent the normally available hardware buttons on a physical phone (figure 3.2) .

The emulator can be set up to use different resolutions and different versions of Android, allowing the developer to verify software compatibility. (Android, 2010)

Figure 3.2: The Android emulator

3.3 Code convention

Code convention is a huge subject of its own, and can be divided into subcategories such as style, language and programming practices. When developing large software systems it is important to follow a common code style convention. A main reason for this is to make the code easier to understand for developers that might work on the system later on. This involves choosing suitable names for variables and classes, as well as making similar choices of code constructions.

Android is an open source project, and a set of rules have been created to keep a common style between developers. These rules are intended for contributors to the android platform itself. Even though they are not a requirement for application development, they are still well thought out and adapted to the Android environment. For this reason these rules were used as guidelines for this project as well. Some of these could be seen as common sense while others add extra readability beyond what is common in Java programming.

The following paragraphs describe some of the important style conventions that were used, and how they differ from the android contributor rules

Javadoc and comments

Javadoc comments were continuously added to the code, using Java's standard format as suggested by Android. Non-javadoc comments were used as well to clearify the code and increase maintainability.

Short methods

Long methods were broken down into shorter ones to increase readability where possible. One good example of this is the method onDraw() (figure 3.3) in the class GameView. This calls several submethods 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 3.3: Pseudocode for the method onDraw()

Fields

Fields should either be at the top of the file or immediately before the methods that use them, according to the Android development rules. All fields in this project were declared at the top of the file. Java code conventions recommend field declaration in the following order: "First the public class variables, then the protected, then package level (no access modifier), and then the private." (**?**).

Limit the scope of local variables

Local variables were initialized on the same line as they are declared if possible, and also as close as possible to where they are actually used, as suggested by Android.

Indentation

The rules recommend using four spaces instead of tabulations. However, since this project was made entirely in Eclipse, normal tabs were used instead. This is the de-

fault way that Eclipce handles indentation and it is facilitated by using auto-indenting options.

Field names and other names

In the project the following field name rules were followed strictly:

- Non-public, non-static field names start with m.
- Static field names start with s.
- Other fields start with a lower case letter.
- Public static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

Additionally, the field names were carefully chosen so that their purpose was clear, and use of ambiguous names or shortenings was avoided.

TODO annotation

The TODO annotation was used during the coding process, to mark unfinished sections in the code. Eclipse automatically recognizes the TODO-comments, making it easier for the developer to find them by marking their locations on the scrollbar while browsing the code.

Logging

Logging was used for debugging purposes. The log method made it possible to spot non-functional sections in the code. Debug messages were marked as verbose, to ensure that they were only compiled in debug mode. In that way they would not affect the performance of the game.

3.4 Git - A version control system

When developing software in teams, a system to manage and synchronize the source code is needed. It ensures that everyone in the project is working on the latest version of the system. Git was used in this project to fulfill this purpose.

When working with Git each developer has one local repository on their computer. Changes made to the code are then copied between the other developers local repositories. This does not force the users to have a dedicated server to store a central repository. Instead Git users are free to store their repositories anywhere. (Git, 2010)

Since Git allows the project to be stored locally on every developer's computer, work can be done on the implementation despite lack of internet access. Any system failure will affect only one individual, providing the project with an increased tolerance to any system breakdowns.

4 The resulting game: Eskimo Tower Defense

4.1 Gameplay

The goal of the game is finish all the tracks on the progression map. To finish a track the player must survive several waves of mobs that try to escape land and jump into the ocean. Each time a mob reaches the ocean the player loses a life. If the player's total life reaches zero, the track is lost. To prevent this from happening the player must build towers that shoot at the mobs.

4.1.1 Progression map

The progression map visualizes the player's progress in the game and lets him choose which track he wants to play. The progression 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 Eskimo Tower Defense it depicts a route from the North Pole to The Sun.

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 player's current progression is visualized with icons. A green icon indicates that the track has been completed. A blue icon indicates that the track is not yet completed, but available to play. A red icon indicates that the track is not yet unlocked. There are currently five tracks called: "North Pole", "Ice Flow", "I see green", "Almost there" and "The sun".

If the player touches one of the tracks, a blue water splash is displayed on top of that track to indicate that it was selected. A menu including the name and current highscore for that track is also displayed. It will offer the options "Start Level" and "Cancel" (figure 4.1).

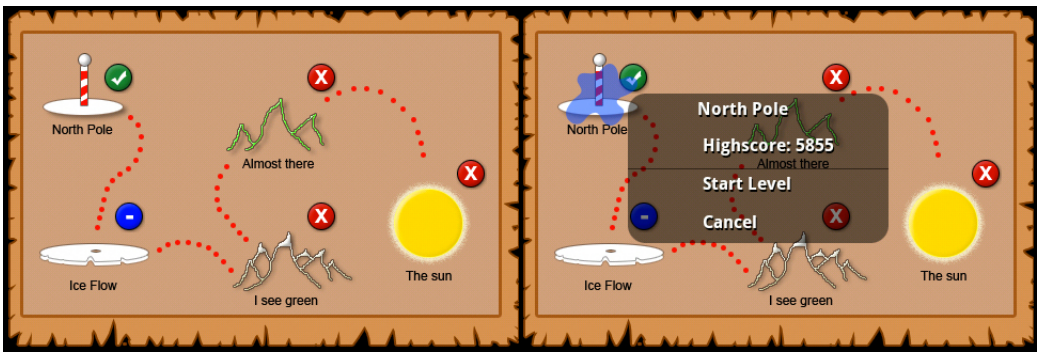

Figure 4.1: The progression map

4.1.2 Game field

Once a track is started from the progression map, the game field is shown. A background for the specified track is displayed. The player is given some time to accustomize himself to the interface, and start building towers before the first wave of mobs enter the game field.

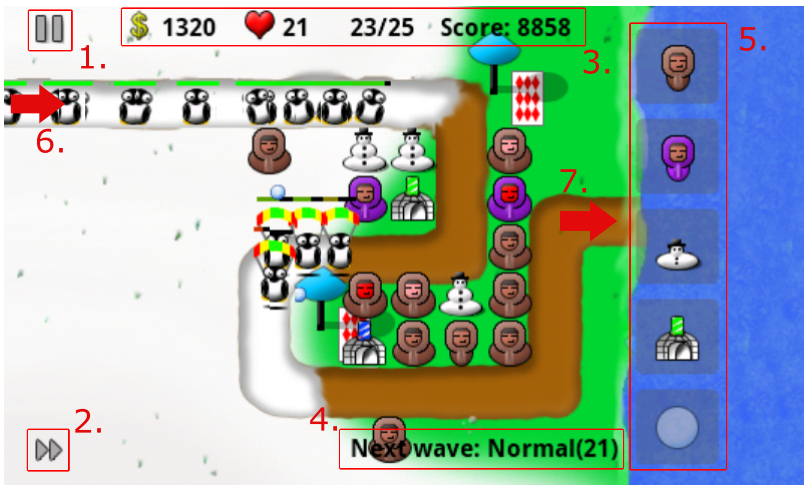

Figure 4.2: The game field

Figure 4.2 shows a screenshot of the game field. Points of interest are marked with numbers:
1. The pause button: Pauses the game.
2. Fast forward button: Triples the speed of the game.
3. Player statistics: Money, number of lives, wave number and score.
4. Countdown for the next wave of mobs. Includes mob type.
5. Buttons for buying new towers. Towers are dragged to the game field.
6. First checkpoint for new mobs.
7. Last checkpoint for mobs on the game field.

4.1.3 Mobs

The word mob is short for mobile and is traditionally used to describe hostile units in video games (Michael Wolff, 2009). In Tower Defense games the mobs are the units that try to cross the game field.

In Eskimo Tower Defense, the mob types are distinguished graphically as different types of cold-weather animals. Each mob type has some unique characteristics. In the current implementation of the game there are five mob types:

- **Normal:** A standard type of mob with no special features.
- **Healthy:** A boss that always comes alone and has more health, making it difficult to kill. They move slower than normal mobs, but are resistant to slowing towers, meaning they are only partly affected by slowing effects.
- **Fast:** Moves faster than normal mobs. Slowing towers can be used to decrease their speed.
- **Immune:** Are immune to the slow effect caused by slowing towers.
- **Air:** A mob type that flies. Splash towers can not damage them at all and basic towers inflict reduced damage to them. Air towers are the most effective weapon against them.

The mobs enter the game field in waves. A wave consists of one or more mobs. All the mobs in one wave are of the same mob type. The mobs enter the game one by one and walk along the path in a line. When the last mob of a wave has entered the game field, there is a small delay before the next wave arrives.

The maximum health of the mobs is increased for each wave. It is up to the player to build towers that kill the mobs before they reach the ocean. If a mob reaches the ocean, the player loses one life.

4.1.4 Towers

Towers are immobile units that the player can build and upgrade for in-game money. By shooting projectiles, they try to stop mobs from reaching the ocean. In Eskimo Tower Defense, the towers are graphically represented by two different Eskimos, a snowman and an igloo.

The different tower types are:

- **Basic tower:** A standard tower type with good range and damage
- **Splash tower:** A tower type with slow fire rate and small range. It shoots a projectile that hits the ground and causes damage to all mobs within its blast radius.
- **Slowing tower:** This tower type is used to slow mobs down temporarily. Its range is fairly small compared to the other towers. It will not shoot at mobs that are already slowed.
- **Air tower:** With big range and damage, this tower type is only able to shoot flying mobs.

Upgrading a tower increases its power in various ways. In some cases, this is more effective than buying new towers.

4.1.5 Projectiles

Towers inflict damage by launching projectiles toward the mobs. Each tower type has a corresponding type of projectile with specific characteristics.

The projectile types are:

- **Basic projectile:** Launched by basic towers. This projectile simply inflicts damage to its target mob without any side effects.
- **Slowing projectile:** Launched by slowing towers. Apart from inflicting a small amount of damage to its target, this projectile also causes a slowing effect when reaching its target. Some mobs are affected less by this effect, some are even immune to it.
- **Splash projectile:** Launched by splash towers. It lands on the ground where the target mob was at the time of launch. When hitting the ground, it explodes and inflicts damage to all nearby mobs.
- **Air projectile:** Launched by air towers. This works exactly like a basic projectile, except it is only shot at flying mobs.

Basic, slowing and air projectiles all lock on to their target mob, changing direction when the mob moves. Because of this, these projectiles always hit their target. The splash projectile does not lock to a target, and there is a risk that it will not hit any targets at all.

4.1.6 Snowball

Several new game mechanics are made possible by accessing the accelerometer. Eskimo Tower Defense features the ability to create a snowball that is controlled by the accelerometer sensor. The snowball is modeled to emulate a real-world snowball, which moves according to the laws of physics. Leaning the mobile device in one direction causes the snowball to accelerate in that direction. When the snowball rolls over mobs on the game field, they take damage equal to 7% of their current health.

Like a real-world snowball, the in-game snowball also melts. This happens slowly over time, and also when the snowball runs over mobs. The more it melts the smaller it gets, eventually disappearing completely.

The player gains access to snowballs depending on his current score. Every time 4000 points have been added to the score, an additional snowball is available. This results in the player being able to use the snowball two to three times per track. No cost is withdrawn when a snowball is used, the snowball is meant to be a reward with no negative side effects.

4.2 Code structure

At the first stage of development of this project, only one activity was used. The game launched directly into the game field, which was a canvas showing a bitmap image representing the ground. This worked well while experimenting with Android graphics. However, when menu development started a few weeks later, it became clear that more activities would be needed. A set of activities was created, where each activity represents a different screen in the game.

The first activity created is called SplashActivity. This is used to display a loading screen for a few seconds. When the loading screen has been shown, a new activity called Menu is created. The SplashActivity is then finished and closed. The Menu activity is not finished every time a menu item is clicked. Instead it remains alive in the background, waiting to be shown after overlying activities are closed.

From the main menu, four different activities can be started: MenuGame, MenuHelp, MenuOptions and MenuCredits. These activities, together with the activity Menu, are the foundation of the game. Figure 4.3 visualizes the flow between different activities:

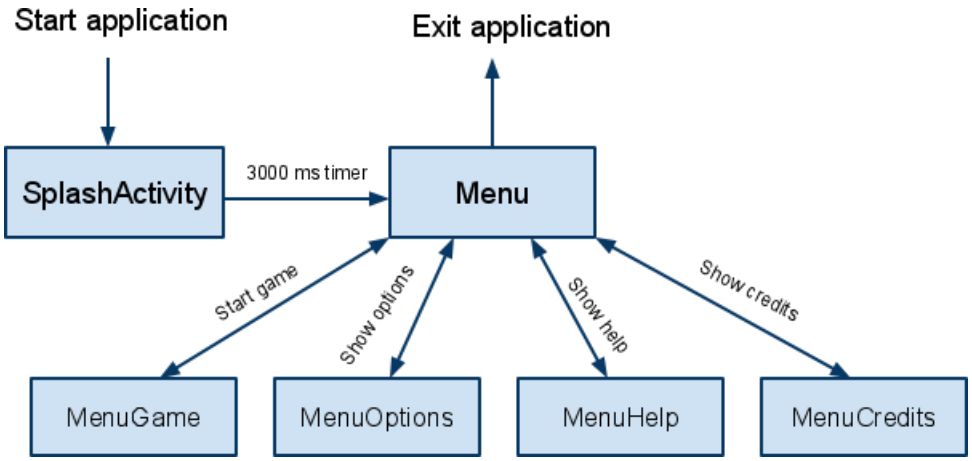

Figure 4.3: Flowchart of activities in Eskimo Tower Defense

Arrows in figure 4.3 indicate a transition between activities. All transitions except the one between SplashActivity and Menu are initiated by user actions, using in-game menu buttons. All navigation is done through in-game menus to give the user a more immersive experience.

The actual game is launched from the MenuGame activity. Two different View classes are used to display either the game field or the progression map. Originally, only the game field View was shown in MenuGame. The progression map was implemented at a later stage of development, and changing the View of MenuGame seemed like the most simple solution. With this solution the transition from the progression map to the game field is performed by the different Views themselves (e.g. when a track is started from the progression map, the game field View is set as the visible View in MenuGame).

The classes MobFactory, Path and Highscore are used to translate XML-files containing information about the tracks, to data stored in lists and arrays. This data is later read from other classes, such as GameView and GameModel.

After selecting "Start" from the main menu and selecting a track on the progression map, the game starts. The game is managed and drawn by the GameView class. This class handles graphics, sound, the game thread, user input, sensor-initiated events like the ones originating from the accelerometer and the different game states (PAUSED, RUNNING, GAMEOVER, WIN).

GameView does not contain any data about the units in the game. Instead, it contains a reference to GameModel which keeps track of all the data related to current states of the units in the game. The GameView's task is to translate the data of the game into bitmaps drawn onto the screen, and to handle user input.

In the constructor, the GameView initializes the GameThread class. This class is a thread that keeps the game running. It continuously invokes the updateModel(), up-

dateSounds() and onDraw() methods in GameModel. This updates the GameModel and the sound states, and draws the screen according to the data in GameModel. The GameThread also includes methods for handling the thread itself, for example a method to terminate the thread when exiting the game.

```
while (thread is set to run by GameView) {
clear canvas
    try {
        initialize and lock canvas with new content

synchronized {
GamePanel.updateModel();
            GamePanel.updateSounds();
            GamePanel.onDraw(canvas);
        }
    } catch (thread is interrupted) {
        do nothing
    } finally {
        if (canvas exists) {
            unlock canvas
        }
    }
}
```

Figure 4.4: Caption for GameThread

The code in figure 4.4 is placed in the run() method of the GameThread. The while loop is ran until it is explicitly stopped from the GameView. The three method calls updateModel(), updateSounds() and onDraw() are placed inside a synchronized statement. This blocks the rest of the program to avoid interference from user input or other threads while a frame of the game is calculated and drawn to the screen.

The purpose of updateModel() is to update the model. The model is represented by the class GameModel that contains lists of all objects on the screen, player statistics, the path and other properties of the game. The updateModel() method performs all the calculations and algorithms that make the game work. It updates the positions of all units on the screen, creates new objects and sets the score according to what happens.

The GameView contains an extra initializing method called startTrack(), which is called from the constructor of GameView. Since the track can be restarted from within the game there must be a method other than the constructor to reset all values, so the game can be played again. If all this code would have been placed in the constructor, the game would have to reload the entire GameView to restart, now it only has to call startTrack().

4.3 Handling user input

Users have different means of interacting with the game. Touchscreen input is a very intuitive method of interaction on modern mobile devices, and it is used for all major user input in Eskimo Tower Defense. The physical interface of the mobile devices is also utilized to some extent, but users are not required to use the physical buttons when playing the game. A new approach to user input in a game of the Tower Defense genre is the use of accelerometer sensors. In Eskimo Tower Defense, events from accelerometer sensors are included in the gameplay.

4.3.1 Touchscreen

One of the more important functions of GameView is the method that handles touch events; onTouchEvent(). This method is called automatically by the system every time the user is touching, holding or releasing his fingers from the screen.

The method does different things depending on which state the game is in, what type of touch event is performed and where on the screen the event was generated. There are four game states: running, paused, game over and victory.

```
if (touch event coordinates matches the pause button coordinates) {
    change game state to paused
}
```

Figure 4.5: Caption for in-game pause menu codesnippet...

The statement in figure 4.5 compares the coordinate of where the touch event occurred to the location of the pause button. If the statement is true, the pause button is pressed, and the state is changed to the paused state.

If a track is lost, the GameView displays an in-game menu with the options "Restart", "Go to map" and "Exit". If the user chooses "Exit" (by touching and then lifting a finger from that button) the method onTouchEvent() is called. The method checks what state the game is in, and finds that it is in the game over state. It then checks what type of event occurred and finds that the event was of type ACTION_UP (a finger is lifted from the screen). Finally it compares the coordinates to the positions of the different menu options. Since the coordinate matches the exit button the method stops the game thread and exits the game (see figure 4.6).

Depending on which state the game is currently in, different coordinates of the screen are checked. Game states are also connected to the onDraw() and updateModel() methods. If the game is in the paused state, updateModel() stops updating the game objects and the onDraw() draws the in-game pause menu.

```
public boolean onTouchEvent() {

    // code

    if (in game over state) {
        if (touch event is action up) {
            if (touch event coordinate matches exit button coordinates) {
                stop the game thread and exit the game
            }
            else if (matches coordinates for other buttons) ...
        }
        else if (other type of touch event) ...
    }
    else if (in other state) ...

// more code
}
```

Figure 4.6: Partial structure of the touch event method

4.3.2 Physical buttons

Every Android phone has a number of physical buttons. As a standard most phones have physical buttons to raise the volume, lower the volume, go back, open a menu and turn the device on or off (see figure 4.7). Some phone models also have physical keyboards. These buttons are handled in the GameView class by the method onKeyDown(). The KeyEvent contains information about which button was pressed and this is checked with a case statement in the method (figure 4.8). It is possible to assign other actions to the buttons than normally intended. For example the volume down button could be assigned to pause the game.

Figure 4.7: Hardware buttons on a HTC Hero

```
switch (keyCode) {
    case KeyEvent.KEYCODE_MENU:
        Handle hardware menu button
    break;
    case KeyEvent.KEYCODE_BACK:
        Handle hardware "back" button
    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;
}
```

Figure 4.8: Caption

In this game both the menu button and the back button pause the game, causing the pause menu to appear by changing the game state to paused. The buttons for raising and lowering the volume are used to control the volume of the in-game sound. Since different phone models have different buttons, only the most common buttons are handled in Eskimo Tower Defense. It is also possible to navigate through the whole game by only using the touchscreen.

4.3.3 Accelerometer

The Android operating system provides access to the sensors of the phone. This was utilized during the development of Eskimo Tower Defense to access the accelerometer. The pseudo code in figure 4.9 shows how the sensors are prepared for access.

```
Access the sensor service of the system

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

Figure 4.9: Caption

An object of the SensorManager class provides access to all available sensors on the device. This object is received by invoking getSystemService() on the running Activity, with Context.SENSOR_SERVICE as an argument. However, the number of sensors can differ a lot between different phone models. This must be handled whenever sensors are used. Otherwise, if a non-existing sensor is accessed, exceptions may occur and cause runtime errors.

The method getSensorList() of the SensorManager returns a list of all available sensors of a given type. To determine whether an accelerometer sensor is available, this method is invoked with the argument Sensor.TYPE_ACCELEROMETER. If the returned list is not empty, it contains at least one accelerometer sensor, and using that sensor is considered safe. The last piece of the code snippet in figure 4.9 registers a SensorEventListener to the accelerometer. This makes the SensorManager generate events that are caught and handled by the listener object.

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

Figure 4.10: Caption

The SensorEventListener interface has two methods that must be implemented: onAccuracyChanged() and onSensorChanged() (figure 4.10). According to the Android API (Android, 2010), onAccuracyChanged() is invoked whenever the accuracy of a specific sensor is changed. However, this is not something that is considered in our software. onSensorChanged() is the relevant method in Eskimo Tower Defense. It is invoked every time the sensor's values are changed, which happens every time the acceleration on the phone is changed. If game objects would be instantly updated when catching the events, performance would become an issue due to the amount of generated events. To avoid this from happening, only the last event is used when updating the game model.

4.4 Graphics

Visualizing the game consists of managing several graphical areas, including XML layout, drawing the maps, towers, mobs, projectiles, animations, tooltips and the towers being dragged across the screen while placing them.

4.4.1 XML layout

Since the class MenuGame is where the progression map is drawn, it does not have a predefined layout. Instead of getting the layout from an XML file, View objects are created and attached to MenuGame during runtime. This class uses canvas, which will be further explained in the Draw section, to show images on the screen.

The main menu of the game and all its buttons have been positioned using the XML editor. The customization of the buttons has been done in the XML editor but drawn

using Adobe Photoshop. As shown in figure xx, the main menu contains a Relative Layout which contains a Table Layout with five buttons: StartGame, Help, Options, Credits and Exit.

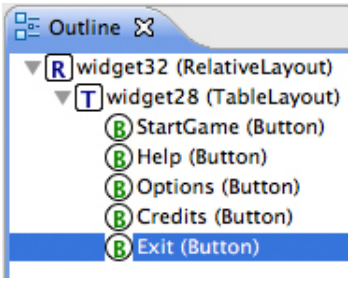

Figure 4.11: Outline view of the layout editor in eclipse

Different properties can be set for the elements of the layout. For example, the Relative Layout has a background image set in the property window. Table Layout has different alignment settings and the buttons have a Background property which contains a reference to a new XML file. This allows different states on the buttons, and the states indicates if the user is hovering over, presses or releases the button.

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

Figure 4.12: Property window of the layout editor in Eclipse.

The XML-file (exit_button.xml), which is referenced from the Exit button's Background property in the main menu, contains the following XML code:

```
<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 4.13: Caption for exit button

Depending on the state of the button, different images are shown.

Figure 4.14: The exit button. To the left, not pressed. To the right, pressed.

4.4.2 Draw

The graphics are handled by the GameView class, which corresponds to both view and controller in the model-view-controller pattern. GameView extends SurfaceView and implements SurfaceHolder.callback() which is called from the main game thread (GameThread). This is done to provide synchronization with the drawing and all values in the game that is being updated.

In the GameView class there is also a method called fillBitmapCache(), which is invoked when the class is created. This method stores all the images that will be used into a cache. This is done to prevent performance issues to occure due to excessive access to these resources. The cache is implemented with a HashMap using the generated resource identifiers as keys. This ensures images are only loaded once.

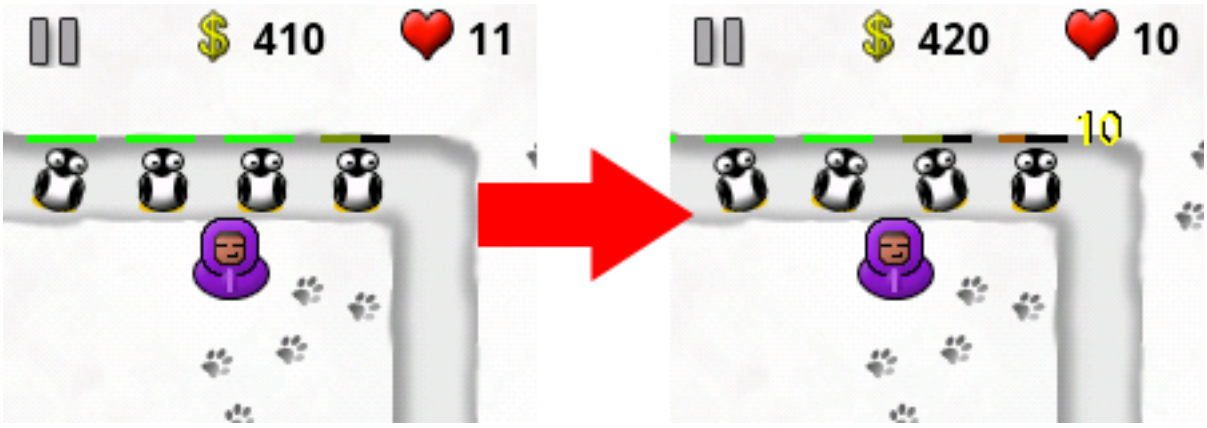

Figure 4.15: Screenshot of a mob dying.

When a mob dies, it is removed from the list of living mobs in the class GameModel and is added to the list of dead mobs called mShowRewardForMob. All of this happens in updateModel() which updates GameModel. The methods updateModel() and onDraw() are frequently called from the game thread. This keeps all values and the canvas up to date.

The method drawRewardAfterDeadMob loops through all the dead mobs and draws the text with the amount of money you get for killing that mob. The method uses the x- and y-coordinate from the dead mob's last position to know where to draw the text. The text moves upwards by increasing the y-coordinate 12 frames before the dead mob is removed from the list and the method completes. As shown in figure XX.

```
private void drawRewardsAfterDeadMob() {
    for (every dead mob) {

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

        if (the reward frame variable is higher than 12) {
            remove the dead mob from the list
remove the reward graphic
        }
    }
}
```

Figure 4.16: Example of how to draw graphic to the screen

4.4.3 Animation

Animating in Canvas is done by switching bitmaps with regular intervals. One of the animations is the water splashing at the end of the map. When a mob reaches the final checkpoint the water splashes up in the air and falls back down again in a smooth manner. This is achieved by showing four images in a sequence.

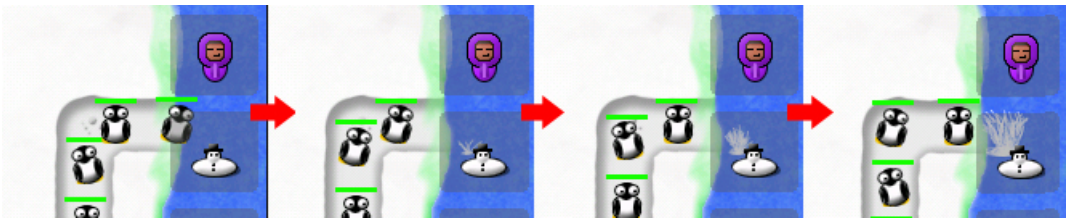

Figure 4.17: Screenshot of water splash animation

The code in figure xx (below) shows how animation is solved in practice. A variable keeps track of how many times the animation has been showed, and is used to determine which image will be drawn to the screen. The water splash animation runs over 25 frames, showing the four different splash images in a sequence.

```
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 4.18: Caption for draw water splash..

4.4.4 Menus

Apart from the main menu that is defined in a XML-file, in-game menus are also implemented and updated in the onDraw() method. There are three different types of in-game menus: The pause-, defeat- and victory-menu. When the pause button on the top left corner on the screen is pressed, the game is paused and the pause menu appears. The game can also be paused when the back button on the device is pressed.

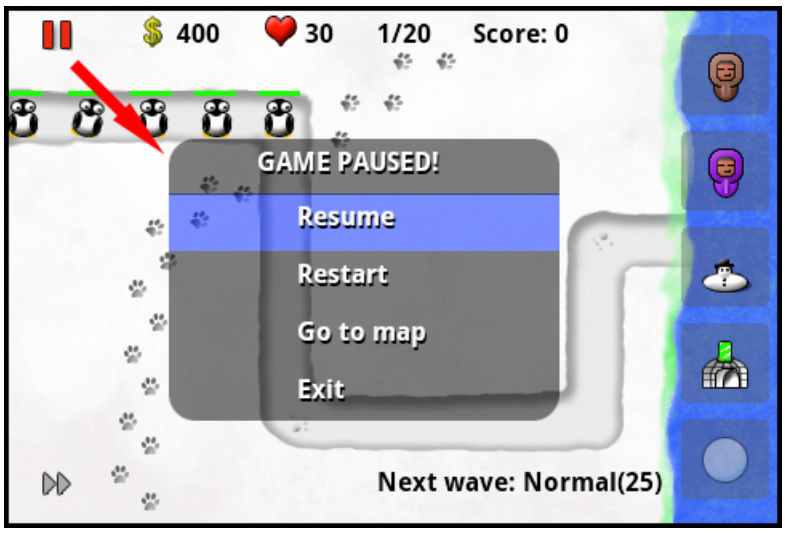

Figure 4.19: Screenshot of in-game pause menu.

The menu shown in figure xx contains five transparent images with text on top of them. There are also versions of the images behind "Resume", "Restart", "Go to map" and "Exit" that have a blue background. These blue images are used as highlighting when the user holds his finger on the button. The user can also drag or swipe his finger over the buttons. The buttons are highlightened depending on the current position of the user's finger. It is only when the finger is released from the screen that the button is logically pressed. Handling user input this way prevents faulty user input, making the game more user-friendly.

4.4.5 Tower placement

Building towers is done by touching one of the corresponding buttons on the right side of the screen, and then dragging the tower to the game field. When the player releases his finger, the tower is built and money is withdrawn. Every tower is taking up 32x32 pixels of the screen and the game field is divided into a grid of squares of 16x16 pixels. There is no grid painted on the screen. Instead of showing the grid visually, the towers will snap to the grid. The player cannot put towers wherever he wants. It is not possible

to build towers on the path, on existing towers, on the pause button or on the fast forward button. While the towers are being dragged, these locations are marked with a red color. A green circle is shown around the tower indicating its firing range while dragging it to the game field. This circle is colored red if the tower cannot be built on the current position, doing so providing instant feedback to the user.

If one of the tower buttons is pressed and held down, a tooltip with information about that tower is displayed. As soon as the tower is being dragged to the game field, the tooltip is removed and a image of the tower is shown. The image of the tower is not located directly under the finger of the user. It would be harder to see the tower during such circumstances. Instead the image is shown some pixels to left of the user's finger. A screenshot of this feature is shown below in figure xx.

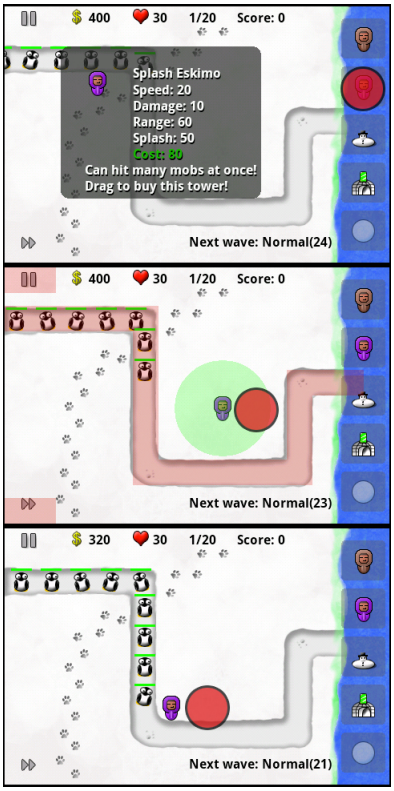

Figure 4.20: Screenshots of a tower being built. The red circle shows where the screen is touched.

4.4.6 Tower upgrade

Upgrading a tower requires the user to press a tower on the game field. A tooltip will then be displayed with information about the tower's current level and the attributes for its next level. Two buttons are available, one button to sell and another button to

upgrade the tower. If the player has enough money to upgrade, the upgrade button is green and if not, the button is red. The colored text indicates how the attributes will change when the tower is upgraded.

When the upgrade tooltip is shown, the game will continue to run in the background. This state is not one of the four states that was described earlier in the Menus section. It is a smaller state in STATE_RUNNING which allows the toolbox to be displayed and the game to be running. The green circle around the tower is also displayed to make the player certain about that he pressed the correct tower.

If the sell button is pressed, the tooltip will be removed and the tower along with it. The upgrade button will upgrade the tower, changing its attributes and image.

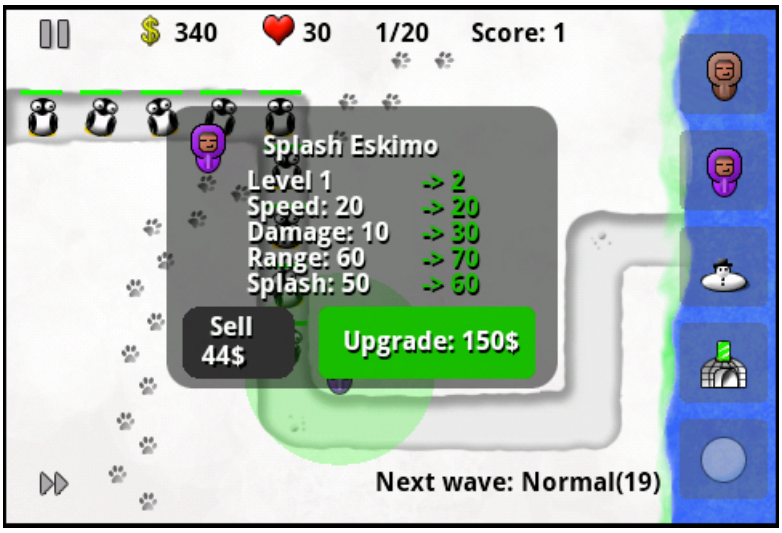

Figure 4.21: Screenshot of the upgrade window

There are four different types of towers and they all have four different levels of power. The image of the towers represents their current level:

4.4.7 Mobs

There are five different mob types in the game: Penguin, Bear, Polar Bear, Walrus and Flying Penguin (see Figure XX). Some of the mobs are animated. This is achieved in the same way as the water splash described in the Animation section. A health bar is drawn above the mobs to show the health of each individual mob. This allows the player to get a more detailed idea of how the mobs health is distributed.

The health bar is not a bitmap but rather two rectangles drawn on the canvas; one for the black background and one for the health (see figure xx). The health-rectangle's length and color is calculated by the health of the mob. The less health a mob has the

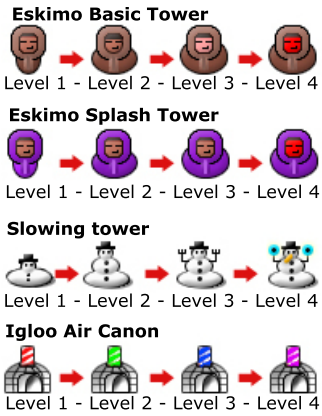

Figure 4.22: All towers of the Eskimo Tower Defense

Figure 4.23: The different mobs of the game

shorter and redder the bar gets. With full health the health-rectangle is green. The ratio of how much health the mob has is calculated by the formula found in figure xx.

Figure 4.24: A mob with a health bar

4.5 Units

There are four different unit types in the game: mobs, towers, projectiles and snowballs. All of these classes extend the abstract class Unit. This class contains the methods and variables that the units have in common. A Unit object contains three instance variables that are set upon creation; height, width and coordinate.

Coordinate is a helper class created to represent and manipulate the coordinates of a unit. An instance of the Coordinate class has two instance variables; x-position and y-position. To make handling of coordinates easier, there is a static method called getAngle() that calculates the angle between two coordinates that are given as parameters.

```
(255 * (double)m.getHealth() / (double)m.getMaxHealth())
```

Figure 4.25: Caption

The method getDistance() returns the distance between two coordinates. The algorithm is an implementation of the Pythagorean theorem (figure xx).

```
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 4.26: Caption

4.5.1 Towers

Tower is an abstract class, that is extended by the four different tower type classes. The Tower constructor takes a coordinate as argument. The rest of the properties of a tower are specific for each tower type and therefore not set by the Tower superclass.

Some of the methods of the Tower class are abstract. These methods are implemented in the different subclasses. These methods are setImageByLevel(), getUpgradeCost(), shoot(), createProjectile() and upgrade().

Since towers are stationary units, they have no method that updates their position. Instead, they have a method called tryToShoot() that is invoked everytime the updateModel() method in GameView is invoked. This method returns a new Projectile object if the tower is allowed to shoot. Towers are allowed to shoot if they are not on cooldown, and have a valid mob within their range. Some tower types have restrictions for what mob types they are allowed to target. When a tower creates a projectile, the tower is automatically set on cooldown for a specific time.

If the tower is allowed to shoot it calls the method shoot(). Basically, the shoot() method iterates over all mobs on the map and to find mobs within shooting range. Mobs that have walked a longer distance on the path are prioritized higher when the tower chooses its target. When a target has been chosen, the method creates and returns a projectile aimed at that target.

The tower is upgraded by invoking the method upgrade(). The upgrade() method increases the level of the tower by one. It then fetches the new attributes from an array of predefined values, using the level number as index.

4.5.2 Mobs

Each mob is represented by an object of the Mob class. The implementation of mobs differ from that of towers. Instead of using subclasses for each mob type, only one class is used. The mob type is set in the constructor, which is only invoked by MobFactory. The most important attributes of the Mob class are shown below.

- **mMaxHealth:** The maximum health of the mob
- **mHealth:** Current health of the mob. Never bigger then mMaxHealth
- **mAngle:** Current angle toward the next checkpoint
- **mSpeed:** Movement speed of the mob
- **mRevard:** Money rewarded for killing the mob. Derived from the maximum health
- **mType:** Mob type (normal, air, fast, immune or healthy)
- **mDistanceWalked:** Keeps track of how far the mob has walked
- **mobImage:** A reference to the image to be drawn on the map
- **mPath:** The path that the mob follows across the map
- **mCheckpoint:** The index of the current checkpoint the mob is walking toward

Mob contains a method called updatePosition(), which moves the mob according to its speed and its current angle. If the mob has reached its current checkpoint, the method also updates the angle to make it point toward the next checkpoint of the path. The updatePosition() method is invoked from updateModel() in the class GameView. When the last checkpoint has been passed by the mob, updatePosition() will return false. This means that the mob has survived walking to the ocean. The mob will be removed from the GameModel, the player will lose one life, and a splash animation will be displayed.

Mob creation using MobFactory

The initiation of mob waves is handled by the singleton class MobFactory. A singleton is a class that only has one instance.***källa***

MobFactory is initiated when the player chooses a track to play from the progression map. When a new object of the class GameView is instantiated, the method startTrack() is invoked. This method invokes getInstance() in MobFactory ensuring that the singleton instance of the class exists. When MobFactory has been initiated, a context must be set using the setContext() method. This is required to be able to reach the resources that contains the XML-file initwaves.

When the method setContext() is invoked, the private method initWaves() is also invoked. This method reads the file initwaves, which is structured as can be seen in figure xx.

```
<resources>
    <array name="mobs_track_1">
        <item>[MobType] [Amount of mobs] [MobHealth]</item>
        ...
    </array>
<array name="mobs_track_2">
...
</resources>
```

Figure 4.27: Caption

The file initwaves contains information about every wave for every track. The XML-code is interpreted by initWaves() as a number of string arrays. Each array represents all waves in one track. Each element in an array holds information about one specific mob wave: the mob type, the number of mobs and the health of these mobs. The information is in the form of a string, and the three different values are separated by the space delimiter.

Starting from track one, initWaves() iterates until a break command is given. This occurs when the counter of the loop exceeds the number of tracks in the XML-file. Each track is identified with a dynamic string that is parsed using the methods getIdentifier() and getStringArray().

The method getStringArray() returns a string array where each element represents an item in the XML-file. A loop iterates over this array and splits each element into three separate strings: mob type, number of mobs and mob health. These strings are handled as shown in the code snippet in figure xx.

When the string has been processed, the correct number of mobs are created and stored in an ArrayList. When all mobs in the wave have been created, the list of mobs is stored in another ArrayList to distinguish it from waves that belong to different tracks.

MobFactory provides a method called getNextMob() that returns the next mob that should enter the game field.

Since getNextMob() is invoked continuously, a delay is implemented to make sure the mobs do not flood the game field completely. This is done by counting the number of times the method is invoked, only returning a new mob when the method has been invoked a certain number of times.

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

if(sType is normal)
    set mob type to normal
else if (sType is air)
    set mob type to air
else if (sType is fast)
    set mob type to fast
else if (sType is healthy)
    set mob type to healthy
else if (sType is immune)
    set mob type to immune

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

Figure 4.28: Caption

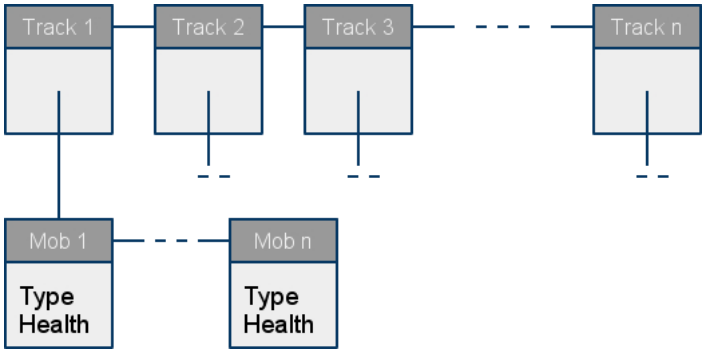

Figure 4.29: Data structure of a wave object

4.5.3 Projectiles

There are four types of projectiles, one for each tower type. They all extend the abstract class Projectile. When a tower shoots at a mob it creates a projectile that moves toward the mob and inflicts damage to it. The projectile has a specific mob as target and is angled toward the position of that mob. For all projectile types except the SplashProjectile this angle is updated every time updatePosition() is invoked to make sure it will not miss its target. The SplashProjectile does not update its angle after being launched, meaning it will land where the mob was when the projectile was launched.

The projectile object has some important properties:

- **mSpeed:** The speed of the projectile

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

Figure 4.30: Caption

- **mDamage:** The amount of damage that the projectile will inflict on its target.
- **mTarget:** A reference to the targeted Mob. Used to calculate the angle.

Projectile contains the method updatePosition(), which updates the projectile's coordinates. The method is invoked by updateModel() in GameView. The new coordinates are calculated based on the old coordinates, the speed and the angle of the projectile.

To determine whether the projectile has reached its target, Projectile has a method called hasCollided(). This method compares the coordinates of the projectile with those of the targeted Mob. If the distance between them is small enough they are considered to have collided, making the method return true. The GameView then removes the projectile from GameModel and the targeted Mob is inflicted with damage.

All projectile types have unique images and methods specific for how they inflict damage. An example of such method is how the SplashProjectile inflicts damage. This is done by looping though all mobs on the map to check their distance from the target coordinate. If they are within a specific range they are damaged. The damage inflicted is lower the further away from the center of the explosion the mob is.

4.5.4 Snowball

The movement of a real snowball is affected by several factors, such as friction and shape. However, in the virtual world of Eskimo Tower Defense it is assumed that none of those factors exist. Snowball objects are given the attributes speed on the x- and y-axis. The updatePosition() method is invoked every time GameModel is updated, with the latest sensor event from the accelerometer as argument. Every SensorEvent has an array of floats that contains the values read from the sensor. Figure xx is a very simplified example of how snowball objects handle accelerometer events.

In this case, the values represent the acceleration the phone is subjected to. If the phone is perfectly leveled the values of the x- and y-axis are 0.0 while the value of the z-axis is close to 9.82 (the gravitational acceleration). These values have the unit m/s^2, but the game field is measured in pixels. Some testing was required to find a meter-to-pixel ratio that gave the snowball the feeling of being controllable yet challenging.

```
public void updatePosition(SensorEvent s) {

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

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

Figure 4.31: Caption

Collision detection is performed in the method getCollidedMobs(), which is invoked from updateModel() in GameView. Because the snowball can collide with several mobs at the same time, all existing mobs are checked for collision. Every mob that has collided with the snowball are added to a list, which is returned from the method. The mobs in the list have their health reduced by 7% by GameView.

Snowball objects have a number of charges that dictate how long they will survive. These charges are decreased both when the snowball collides with mobs and at regular time intervals. The radius of the snowball depends on the amount of charges. The radius is used both for collision detection and the drawing of snowballs on the screen. When the number of charges is reduced to zero, the snowball is removed from GameModel.

4.6 Path

The class Path represents the road that the mobs walk along. Each track has a unique path. Path contains a list of coordinates for all checkpoints of the road. A checkpoint is a point where the direction of the path changes.

The Path in itself is not visible to the user, but the background image of the track contains a visual representation of it. The path is used by the two classes Mob and GameView. Mobs use it when they update their position on the game field. They use the coordinates to calculate in which direction they should move. GameView uses the path to calculate where the user can place towers, since the user is not allowed to place towers on the path.

The Path class is implemented as a singleton, so when the track is changed, the Path object is just reset and reused. The Path object is, just like the MobFactory object, first initialized when the player chooses a track from the progression map. The Path and MobFactory singletons are almost identical in their structure, but they handle different kinds of data.

When the path is initialized, information is read from an XML-file called initpaths. It is structured using one array for every track, where every item represents a checkpoint coordinate. Figure xx shows the structure of initpath.

```
<resources>
    <array name="path_track_1">
        <item>[X coordinate] [Y coordinate]</item>
        ...
    </array>
<array name ="path_track_2">
...
</resources>
```

Figure 4.32: Caption

The initiation contains two nested loops that iterate over every array tag and every item tag. The data is saved in an ArrayList, where every element is an ArrayList of coordinates (Figure xx).

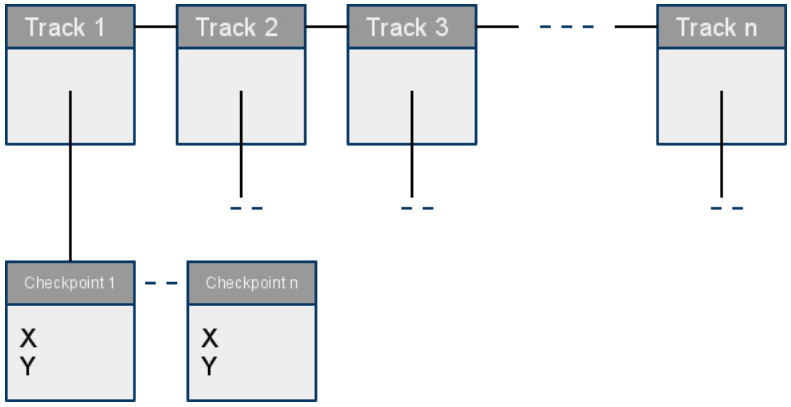

Figure 4.33: Caption

4.7 Money and highscore

When a mob is killed the player is rewarded with money and points. The money is used to buy more towers as the game progresses and can thus establish a stronger defense. If the player fails to kill a mob, not only does he lose a life but he also misses the reward money. This makes it harder to defend against subsequent mobs.

The points are added to the playerś total score for the current track. The rewarded points are calculated based on the start health of the killed mobs, and decreased by a

factor between 1-2 depending on how far the mob has travelled before it is killed (figure xx). This introduces some variation to the high score, ensuring players are rewarded for a more effective defense. With the distance included in the formula the highscore gets bigger if the player kills the mobs early.

```
if (Mob has walked less than 500 pixels) {

The new score is set to:

    Old score + (Mob max health / 10) *
        (the distance the mob has walked / 500 pixels)

} else {
    Old score + (Mob max health / 10) * 0.5
}
```

Figure 4.34: Caption

The highscore is managed by the singleton named Highscore. When first initiated, it tries to read from the text file tddata. If it is the first time the game starts, or if no track has been completed, the file does not exist, and an exception will be thrown. When it is caught, the text file tddata is created (figure xx).

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

Figure 4.35: Caption

The read and write methods are used on several occasions. To reduce the total amount of code, the initializations of BufferedReader and BufferedWriter are done in separate methods.

The method saveCurrentTrackScore() is invoked when a track is completed, saving the score to the text file tddata. The high score file is read in order to display each track's highscore on the progression map. The procedure also provides a way to determine if a certain track has been completed, since the highscore is zero only if the track has never been completed. This is used to unlocked the correct tracks on the progression map.

5 Discussion

This section details several of the design decisions that had to be taken regarding gameplay. It also discusses why other ideas were not implemented. The main goal was for the game to provide a new and fun experience for the user. This meant that a lot of effort was put into making the game exciting. This section also details some of the discussions that were held regarding the concept of the game.

The discussion also provides some comments on how the project could have been managed better, and how certain issues could be handled more effectively.

5.1 Design choices

Deciding what features to implement was an important step in developing a foundation from which a marketable game could be developed. These decisions were based on evaluations of existing solutions already on the market, as well as some solutions that did not already exist on the market.

5.1.1 Fixed path versus maze building

The first important design decision that had to be taken included the movement pattern of the mobs. The Tower Defense genre is divided into two distinctive groups when it comes to mob movement over the map: fixed path and maze building. With a fixed path the mobs move along a pre-defined route on the track and towers can be built along its sides. Maze building games on the other hand, consist of an open field where mobs can move freely. It is up to the player to build obstacles forcing the mobs to take detours toward the exit. The obstacles that are used are in most cases the towers themselves.

While testing different games during the research stage of development, one of the tested games was Robo Defence (Lupid Labs, 2010), which featured maze-style gameplay. Robo Defence is one of the most popular Tower Defense games currently on the Android market. Distinguishing Eskimo Tower Defense from this popular alternative was a big factor in deciding upon the fixed path design.

There are many benefits of using a fixed path model. Using a fixed path makes the game less complex to the player. It is more forgiving when it comes to the placement of

towers: Maze building games can become frustrating if the control is imprecise, because a tower in the wrong place might destroy the entire maze. Another benefit is that it gives the level designer more control over the tracks, making it easier to vary track design by varying the path.

Another reason for using the fixed path solution was that none of the team members had much prior experience of game development. For this reason it was decided to keep the complexity at a minimum. The maze building solution would imply the use of artificial intelligence which would be too time consuming to implement, because of the increase in complexity. The static path design seemed to be the least complicated to use when developing a game for the first time. To assure a unique gameplay the focus was instead directed to other implementations, such as the snowball.

5.1.2 Theme

One of the major subjects of discussion in the group was the theme of the game. Since no one in the team had any experience of graphical programming, the theme was decided late in the project. At first, placeholders were used to test the functionality of the game. This was later changed to a temporary theme to get started with the graphics.

It was decided that the game should have a background story that the theme should be linked to. In an early discussion, a green house effect-theme was discussed. The story would then be that animals from the north pole would flee from their melting environment. A map with that story would then have different themes, for instance polluting factories, melting ice caps and smog.

Finally it was decided to use an Eskimo theme. The background story was the same but instead of the green-house effect the animals were migrating south because they desired to move to a warmer climate. The Eskimo tribes frowns upon this since they depend on the wildlife for food and clothing. The tribes therefor set out to prevent this disaster from occurring. The maps will gradually become greener and warmer when progressing throughout the game. The towers were designed as different Eskimos, the snowman and the igloo canon. The mobs became animals such as penguins, polar bears and walruses.

5.1.3 Projectile algorithm

During the discussion of how the projectiles should work, two different ways of implementing them were suggested. One way was to make them constantly follow their target, changing direction as they moved, like a homing missile. The other approach was to approximate where the mob and projectile would collide and aim in that direction. With this approach, the direction of the projectile does not change after being launched.

Both methods where implemented to see which one had the best performance. It turned out that the homing missile method was optimal. Since the projectiles travel quite fast, most often reaching its target in about three frames, only three corrections of the angle were needed. When using the interception method the angle is only calculated once, however, this approach also requires the calculation of the future position of the mob. The code for the interception equation is shown in figure 5.1.

```
set target coordinate to:
    (getMob().getX() + getMob().getWidth()/2,
getMob().getY()  + getMob().getHeight()/2)

calculate the angle, a2, between the projectile and its target

set angle to:
    (a2 - Math.asin(getMob().getSpeed()/getSpeed()*
Math.sin(Math.PI - a2 + getMob().getAngle())))
```

Figure 5.1: Pseudo code for interception algorithm

The equation for the homing method is shown in figure 5.2.

```
set target coordinate to:
    (mTarget.getX() + mTarget.getWidth()/2,
    mTarget.getY() + mTarget.getHeight()/2)

set the angle of the projectile to:
    the angle between the projectile coordinate and the target coordinate
```

Figure 5.2: Pseudo code for homing algorithm

According to tests performed, the first equation is more than three times as resource-demanding as the second one. This can be explained by the use of Math.sin() and Math.asin() methods which are much more time demanding than a normal multiplying operation CITE"". It was therefore decided to use the homing missile method.

Another issue is how to detect a collision with a mob. Collision detection of a mob and a projectile is done by checking the distance between them. If the distance is smaller than a predefined value the objects are considered to have collided. The problem is which mob to calculate this for; either the target mob of the projectile or any mob coming in its way. The most intuitive approach would be the second one. The problem with this is that the projectile has to check for collision with all the mobs on the map. This can be very resource consuming, especially if there are a lot of mobs and projectiles currently on the map. It was therefore decided only to do collision detection for the target mob. The only real problem with this is that if a projectile misses its original target, it will

not be able to hit another mob. Since we decided to use projectiles acting like homing missiles this would not be a problem.

5.1.4 Waves

A big issue was how the waves were supposed to work. In some Tower Defense games, such as Bloons (Ninja Kiwi, 2010) and Flash Element Tower Defense (David Scott, 2007), the game pauses between every wave, giving the user time to build more towers. This means that the user will have to manually call the next wave. Another approach is to have the waves arrive continuously with a fixed amount of time between them. It was decided to use the delay approach. This was to ensure that the game would keep a steady pace and to remove unnecessary user actions. Since the tracks feature lot of waves this might be burdensome for the player to always have to press a button between each wave.

Since the mobs have varying speed, this may result in having one wave catch up with the previous wave. This is very noticeable during the boss waves. A boss wave consists of one mob with a lot more health than normal mobs, which means they take more effort to kill. Since there is only one mob in boss waves, the countdown for the next wave starts immediately after it has entered. To solve this, the delay after the boss waves was increased.

5.1.5 Unique features

The development of the game called for distinguishing it from other Tower Defense games to make it more desirable for potential customers. Modern Android phones offer many new ways to control applications which could be taken advantage of. The idea was to add an extra dimension to the game using the phone's accelerometer. One of the ideas was to control the towers by tilting the phone. Another idea was that the player would be able to control the range of the towers. If the player would tilt the phone to the left, the towers' range would be increased in that direction. Another idea was to control the speed of the mobs. When tilting to the left, it would create a slope to the left making the mobs moving left go faster and mobs moving right go slower.

The studies did not include any games that had multiple paths for the mobs. This would be an easy feature to implement that would make it more unique. Another thought was to allow the user to control which way the mobs walk in a crossroad using the accelerometer. The more the phone is tilted in one direction, the higher the probability is that the mobs choose that way.

The idea that eventually was implemented was the snowball. The snowball will roll over the map controlled by tilting the phone, almost like the classical board game Labyrinth. The player can use this snowball to kill mobs when in difficult situations. Some discussion

was had regarding how the snowball would interact with the units on the map. Initially it was designed to kill every mob it touched. This was the easiest way to code it but it made the snowball too good, especially against bosses who had much more health than other mobs. Later, this was changed to dealing damage based on a percentage of the mobs health every frame it touched them. The snowball was also modified to deal less damage to bosses in order to make it less effective against them.

There was also a discussion whether the snowball should have any negative aspects 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 kill mobs very effectively. This would discourage the user from tilting the phone, not using the snowball as it was intended. The snowball was well recieved in several of the post-development interviews.

5.1.6 User interface

User interface is one of the most important areas when working with touchscreen mobile phones or small touchscreens in general. The small screen needs to hold a lot of information. Since the touchscreen is operated with the user's fingers, and some might have bigger fingers than others, it is important that items are not too small. This can make the player irritated when trying to touch the buttons. They should not be too big either because that would result in the game field being too small.

You want to fit as much information as possible on the screen, but you also want to keep it clean for the user. There were many discussions about which buttons would be the most important for the player during the different states of the game. To keep it clean, pressing a button often brings up a menu with several options to choose from.

The game is played with the screen in landscape position, meaning that most users will only use his thumbs to interact. This means that the buttons must be even bigger compared to if the index finger would have been used.

5.1.7 Animations

Animations were not the main priority during the development. At first, the development was focused on having a working game with innovative features. During testing, it became clear that a game with no animations would be pretty boring. To give the player a good experience the game needed a more realistic feeling. The interviews also revealed that sounds and animations were positive for the game experience. Therefor animations were added to the mobs, which made them look like they were wobbling back and forth when they were walking down the path. This small change made the game

look much more dynamic. There is also an animation at the end of the path where the mobs dive into the water.

5.2 Game balance

For a game to be interesting and thus sellable, it must provide just enough challenge to the user. In strategic games like Tower Defense, it is also important that there is room for different types of strategies. Both of these requirements are achieved by balancing the game. This section describes how the different parts of the game were balanced and why.

5.2.1 Towers

Balancing of the towers was done to make sure that no tower was superior to other towers. Being able to finish the game by only building one type of tower would make the game boring and unchallenging. The 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. All maps contain different combinations of mob types, meaning the user has to change his strategy to meet the challenges he faces.

5.2.2 Mob waves

The biggest part of game balance included adjusting the characteristics of mob waves. Each mob awards the player with money when killed. Making the game balanced requires the income to be similar to the cost of building towers. If too much money was rewarded the game would not be challenging and if the reward was too small it would be impossible to finish the game.

Since the maps have different difficulties, we had to manually set the health of each mob wave. The waves of the first map should be easier than the waves of the second map, increasing the difficulty as the player progresses.

5.3 Critical project review

At the end of the project, several points were raised that could have been approached in a different manner. The primary concern was that the developers gained a much greater understanding of the technology behind several of the key concepts. In particular, the way Android uses activities, but also the use of XML and how touch sensitivity was to be implemented. The project could have benefited from having these concepts studied and

analyzed further before work on implementation began. This would have meant that a greater understanding would have been gained which would have positively impacted the results, as well as increasing the efficiency of work done.

The project would have progressed smoother if a more strict project development method was chosen and consistently followed. A discussion was held at the beginning of the project where the decision was made that a model of the project would be designed before implementation. This was to allow the team to focus on making many of the key design decisions early on, instead of having them come up at a later time disrupting work. This modelling prior to implementation was not fully completed, resulting in the same problems it sought to avoid. The amount of problems caused by poor design decisions might theoretically also have been greater because of this.

One problem which could have been handled better was that of the cross platform functionality. Since several Android phones varied in screen size and resolution, we resorted to a compromise when drawing the canvas. This resulted in a slightly misplaced interface on certain phone models. This was handled by drawing a game field that was slightly larger than necessary, which resulted in the buttons being located in a less than optimal position on the larger phones compared to the smaller ones. This can be compared in figure XX.

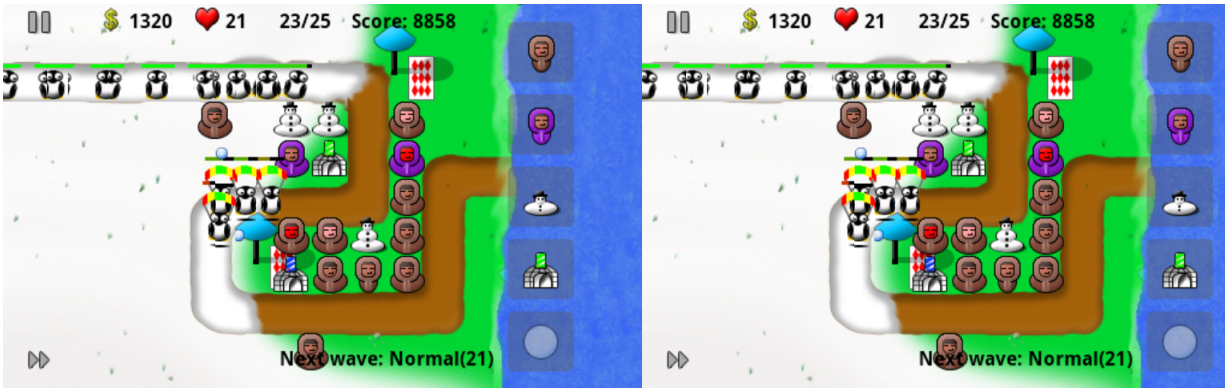

Figure 5.3: To the left: The game on a HTC Desire, To the Right: The game on a HTC Hero

Several of the assumptions that ultimately decided many of the major design choices were somewhat based on results of the pre-development interviews. However, there are a few improvements that could have been made. The interviews were not in a particularly large scope, only detailing a few individuals, meaning that they likely do not represent a majority of the possible user base. The interviews were also conducted at a very early stage and failed to bring up several of the more important questions that were contemplated during the design discussions. Instead, the interviews were mostly focused on general questions which generated general answers without actually providing

a base for several decisions. Design decisions were instead based on some of the most popular Tower Defense games.

A few interviews were also conducted after the development process to get a general idea of how players felt about certain aspects of the game. These interviews were mostly held with friends and family of the developers which might have heavily influenced their comments regarding the game. What the game might have needed was a more extensive pre-development and post-development research regarding the market, including focus groups, as well as further discussion. (Lindlof 2002) CITE

5.4 Future work

The game that resulted from this project is playable in its current state. However, additional features could be added to increase the value of the game. It is also an excellent basis from which a commercial game can be built. This chapter includes several implementations that were thought of but never realized.

5.4.1 Public highscore

To make the game more attractive and addictive, which was part of the purpose of the project, a public highscore could have been implemented. This would allow the users to play not only to complete the game, but also to compete against other players. A server would be needed to be able to upload and store the users' highscores. This was considered to take too much time and was not that important for the overall game experience, therefore it was not implemented.

5.4.2 Cross-platform speed consistency

If the game was ever to have a public highscore, the game would need to be fair. If the game runs faster on phones with more powerful processors it is not fair. In that case, players with slower phones would have an advantage compared to the rest of the players. To solve this the game has to run at the same speed independent of the speed of the phones' processors. There is also a problem with future phones that will have even more powerful processors. These phones would run the game so fast it would be very hard for users to play it. There are a number of well-known methods available to counter this effect. These were excluded from this project in favor of adding features that more directly affect the game experience. This issue has the highest priority of all future work.

5.4.3 Sounds

In the current version of the game the only sounds included are the background music and a sound effect played when the mobs reach the water. According to our last interviews many people wanted sound effects when mobs were killed. According to them, this would increase the addicting factors of the game. Implementing more sound effects was planned but there was no time to add this for all situations in the game.

5.4.4 Achievments

One thing that might increase the game's lifespan is the concept of achievements. This is a form of extra bonuses given to the player for completing various predetermined tasks. Completing these achievements would unlock new functionalities or new maps. These achievements could be relatively hard to complete. They would also give the user something to strive for after completing all the standard maps in the game.

5.4.5 Snowball improvement

The snowball is one of the main parts that separates Eskimo Tower Defense from other Tower Defense games. To further improve the game experience, it is very important that this special weapon is fun and easy to use. If it is not implemented good enough, the player might not want to use it. He would then miss out on one of the things that makes the game unique. Further development of the snowball is therefore an important part to focus on.

Since the snowball is never introduced properly, new players might not notice that it exists. The snowball is one of the more unique parts of the game and it would be very bad if the player never uses it. A message or sound that indicates that the snowball is available would be helpful. The way the snowball is implemented could also be improved. It could for example bounce or even damage towers on the map to make it more challenging to use. The graphics for the snowball could also be improved. In the current version, the snowball is able to roll over the water. One suggestion was that is should sink or melt faster in the water. Instead of killing the mobs, there was an idea to have the snowball stun them. This would stop them from walking for a short moment, giving the towers more time to shoot them.

5.4.6 Additional special weapons

To make our game more unique, additional special weapons could be implemented. One idea was to variate the snowball with other player-controlled weapons. A weapon that would daze or stun the mobs momentary was discussed. This could be graphically

represented as an earthquake for example. The idea of a meteor falling down from the sky was also discussed. These different weapons could be gained by completing different maps or achievements. They could also be available on certain maps where the special weapon is related to the theme of that map.

5.5 Conclusion

A Tower Defense game for Android has successfully been developed during this project. With regards to functionality, all of the major components that make up a Tower Defense game have been implemented, and would only require optimization to form the basis of a marketable game. Concerning the development of innovative game mechanics, the snowball was well recieved in the post-development studies and is a prime example of new user interaction features made possible on the Android platform.

According to our post-development interview the balance of the game is a big part in the feeling of the game. After developing the game our conclusion is that the balancing is a very complicated part, that requires both time and insight into the game structure. When changing one type of balance variable it affects many other parts, requiring extensive testing before a complete game is produced.

Bibliography

Android. Android api. http://developer.android.com/, June 2010.

AndroidLib.com. Robo defense. http://www.androlib.com/android.application.com-magicwach-rdefense-jpjz.aspx, May 2010.

Canalys. Majority of smart phones now have touch screens. http://www.canalys.com/pr/2010/r2010021.html, February 2010.

Eclipse Foundation. Eclipse documentation. http://www.eclipse.org/, May 2010.

Helm R Johnson R Vlissides J Gamma, E. *Design Patterns*. 1995.

Git. Git - fast version control system. http://www.git-scm.com/, June 2010.

Robert Green. Getting started in android game development. http://www.rbgrn.net/content/54-getting-started-android-game-development, 2008.

Robert Green. How to test android performance using fps. http://www.rbgrn.net/content/286-how-to-test-android-performance-using-fps, May 2009.

Lev Grossman. The best inventions of 2007. http://205.188.238.181/time/specials/2007/article/0,28804,1677329_1678542_1677891,00.html, November 2007.

Derek Baker Nathaniel Wice Michael Wolff, Kelly Maloni. *Netgames: your guide to the games people play on the electronic highway*. University of Virginia, 2009.

Sun Developer Network. Code conventions for the java programming language. http://java.sun.com/docs/codeconv/, May 2010.

NinjaKiwi.com. Bloons tower defense 3. http://www.ninjakiwi.com/Games/Tower-Defense/Play/Bloons-Tower-Defense-3.html, May 2010.

David Scott. Flash element td. http://se.t45ol.com/spel/2951/flash-element-td.html, May 2010.

SQLite. Sqlite documentation. http://www.sqlite.org/, June 2010.

Bryan C. Taylor Thomas R. Lindlof. *Qualitative communication research methods*. 2002.