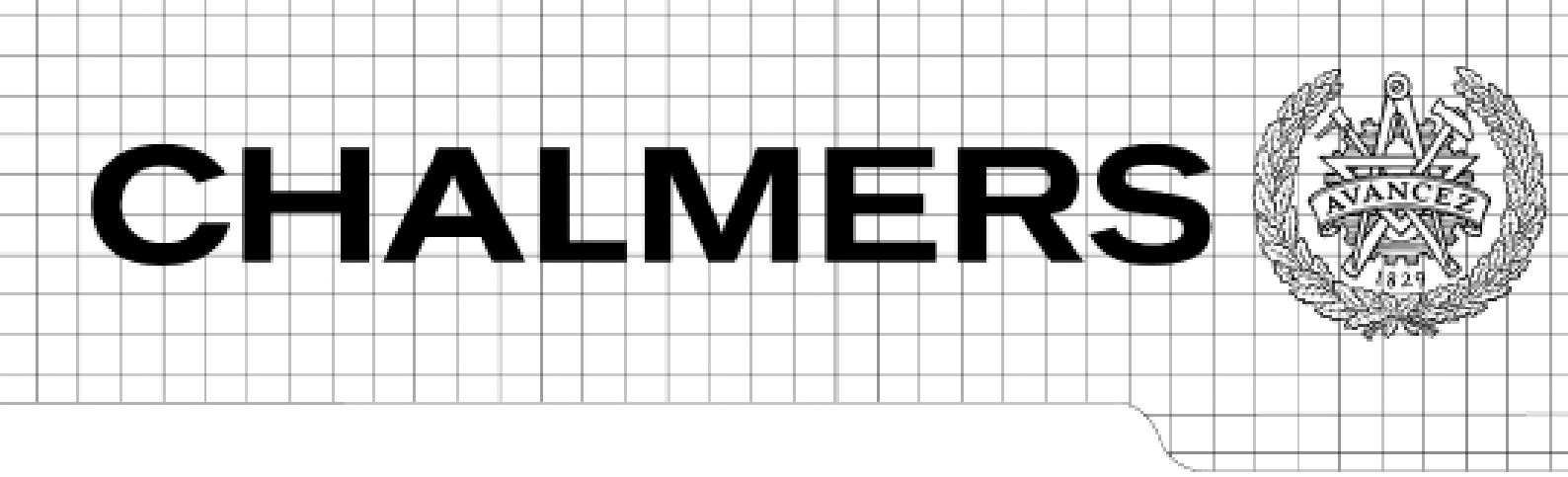

Tower Defense for Android

Bachelor's Thesis
Computer Science and Engineer Programme

Authors:

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

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

Abstract

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

Tower Defense is not a new game concept. There are several games of this type with different themes and gameplay already on the market. In order to provide a solid foundation from which a new Tower Defense game can be developed, many of the existing games were examined thoroughly to find desirable features. In addition to this, interviews with people that play Tower Defense games frequently were conducted, making sure no important features were left out.

The conclusion of the research was that in order to make the game competitive on the Android Market, the gameplay had to be unique. The features of the mobile device were used as tools to achieve this goal. Smartphones primarily provide two features that were taken into consideration when developing Eskimo Tower Defense; the touchscreen and the accelerometer. The touchscreen provides a new interaction method in comparison with similar games on a computer platform. However, this alone did not suffice to distinguish a Tower Defense game on the Android platform. Eskimo Tower Defense utilizes the accelerometer to give players a feeling of being able to control the game, making them more physically involved in the gameplay.

Contents

Vocabulary

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

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

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

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

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

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

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

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

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

1. Introduction

1.1. Background

In the beginning of 2007 a powerful development of smartphones in the mobile phone market was introduced. The introduction of the Iphone by Apple was the starting signal of this rapid development. Since then, several other IT companies have released this type of phone. Most of these smartphones have operating systems from Symbian, RIM, Apple, Microsoft or Google CITE"(Canalys, 2010)". What they all have in common is that it is relatively simple to develop software for them, meaning that anyone with basic programming skills can develop their own applications.

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

The development of a game was chosen because it consists of a lot of the common obstacles encountered during software development. The difficulties of programming a game include drawing of graphics, managing code efficiency as well as implementing a good system for user interaction. A game is also a good place to start when experimenting with new input methods. The concept of Tower Defense was chosen for the fact that there were few if any Tower Defense games that featured the use of an accelerometer in its game design.

A Tower Defense is a common type of game often found as Flash-applications on websites. The general concept is that the games feature creatures, organized into waves, that try to go from point A to point B. The objective of the game is to prevent the creatures from reaching their final destination. This is achieved by constructing towers that fire automatically and intermittently upon creatures that enter their range. The game is often varied by introducing different paths creatures can take; some games allow players to construct obstacles that force the mobs to take a certain path, giving the towers additional time to fire. The games often feature different towers with different purposes, as well as different creatures with varying attributes, weaknesses and strengths.

1.2. 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 for the game, so that further development is facilitated. The number of tracks, tower types, mob types and different sound effects is intentionally small, since focus lies on functionality rather than quantity. Game balance has a huge impact on commercial viability. If people think the game is unbalanced, they might not want to play it. Due to this fact, one of the goals of this project is to make the game feel as balanced as possible. This can be done by altering values and properties of different game objects, creating synergy and making all elements of the game attractive to the user. The game should be easy to play but still provide a challenge for the user.

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

2. Theory

2.1. Android architecture

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

Gaining access to the surface of an Android device requires an implementation of the Activity class. The first describing line in the Android API about activities is "An activity is a single, focused thing that the user can do" CITE(Android, 2010). 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 that is designed for applications running in the background).

As can be seen in figure 2.1, onCreate(), onStart() and onResume() are all invoked as an activity is first created. Several other methods are also invoked 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, when a lot of variables are instantiated in the onStart()-method, memory leaks might occur if they are not set to null in the onStop()-method. Memory leaks can cause the entire device to slow down, which 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 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 being used to

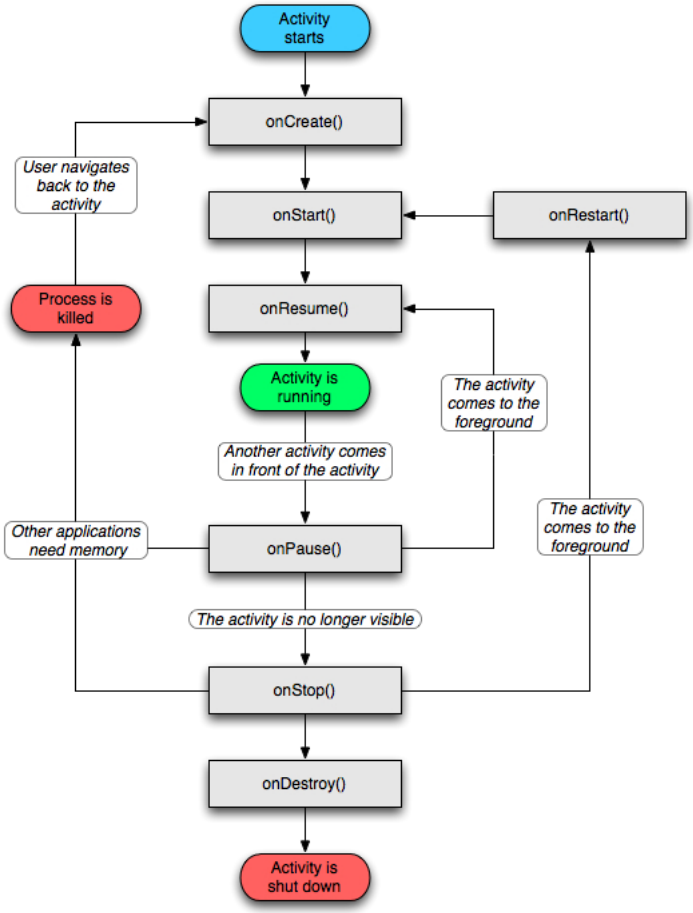

Figure 2.1.: Life cycle of an Activity

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.
CITE (http://developer.android.com/guide/topics/resources/resources-i18n.html)

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 CITE (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 from the file system on a mobile phone; preferences, files, databases and network. CITE(Android 2010)

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

Android also comes with the possibility of using databases for data storage. The type of database available on a Android device is SQLite. CITE(Android 2010) SQLite is a lightweight database engine, built to suit devices with limited memory. It reads and writes to files on the deviceś file system. "A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file." CITE(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 is handled like layers even if it is not exactly 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 the layout. There is a built-in XML-editor in the Android Software Development Kit for Eclipse that you can use to create these layouts. The editor shows a preview of the window that will be shown on the screen of the device (see figure 2.2). Inside that window is where all the graphics are put. The editor is very easy to use as it uses the drag-and-drop concept. There are several layout options such as GridView, ListView and LinearLayout to choose between and 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.

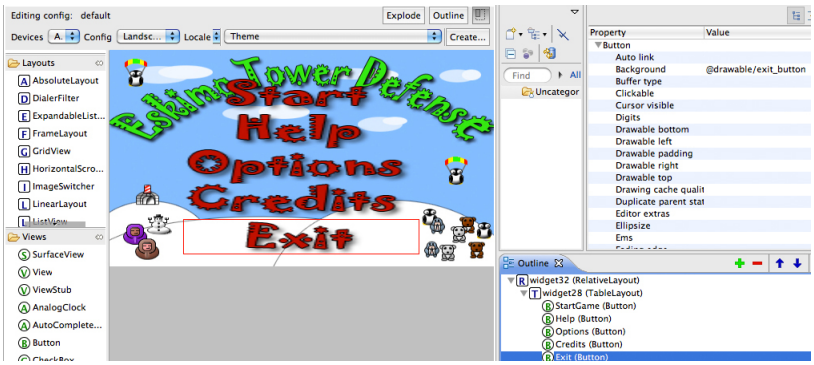

Figure 2.2.: Caption for the xmleditor

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, it is clear that all of the major sound formats are supported (MP3, MIDI, WAVE, Ogg Vorbis and more). Unless playing sounds is the main purpose of the application, it is desirable to use files that are small and do not require lots of memory.

Implementations of using sound in Android applications are done by utilizing the classes MediaPlayer and SoundPool. The SoundPool class should be used for small sounds, sounds that are likely to be repeated a lot. Properties of this class are described in the Android API (Android 2010). 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 in games.

3. Method

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

The development was done in Eclipse IDE with the Android SDK-plugin with the program language Java. Included in Android SDK is an emulator which made it possible to test the 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 be able to make the project easy to maintain, subversion management with Git was used. This increased our ability to share code within the developer team during the development process, and to allow us to work on multiple parts of the project at the same time.

3.1. Interviews

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

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

The interviews provided some insight into peoples thoughts and the statements from the participants were used when making important design decisions. It soon became 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.

3.2. Git - A version control system

When developing software in teams, a system to manage and synchronize the source code is needed. This to ensure everyone in the project is working on the latest version of the system. In this project Git was used for this purpose.

When working with Git each developer included in the project has one local repository on their computer. Changes made to the code is 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 members computer, work can be done on the implementation despite lack of internet access. This also means that any system failure is only going to affect one individual, who could then fetch an updated version of the project from the other members, providing the project with an increased tolerance to any system breakdowns

3.3. Code convention

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

3.3.1. Style conventions

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 are 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 the standard format as specified in by android. Non-javadoc comments were used as often as well to clearify the code and increase maintainability. A comment including copyright info were not added, since this was deemed to not be needed for the project at this stage.

Short methods

Long methods were broken down into shorter ones to increase readability where possible. One good example of this is the method onDraw in the class GameView. 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 3.1.: Caption for onDraw...

Fields

Fields should either be at the top of the file or immediately before the methods that use them, according to the rules. All fields 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." CITE(http://java.sun.com/docs/codeconv/)

Limit the scope of local variables

Local variables should be initialized on the same line as they are declared if possible, and also as close as possible to where they are actually used.

Indentation

The rules recommend using four spaces instead of tabulations. Since this project was made entirely in Eclipse we chose to use normal tabs. This is the default way to handle indentation when using Eclipse and it is facilitated by using auto-indenting options.

Field names and other names

For field names the rules were strictly followed. This means:

- 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 field name 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 extensively during the coding process, to mark unfinished sections in the code. Eclipse automatically recognizes the TODO-comments and makes it even 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 sections in the code that was not functional. Debug messages were marked as verbose, meaning that they were only compiled in debug mode. This to ensure that they would not affect the performance of the game.

3.4. Android development environment

Developing standard Android applications requires the Java Development Kit (JDK) and an Integrated Development Environment (IDE). Google recommends Eclipse with Android Development Tools (ADT) for programming applications to Android. The ADT is a plugin for Eclipse allowing easy creation and management of Android projects. CITE HERE””

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 EE Developers that currently has over 1.2 million downloads. Eclipse IDE is free to use and allows the users to easily create Java applications. (Eclipse Foundation, www.eclipse.org, 2010-04-30)

Figure 3.2 shows a screenshot of a project in Eclipse. The left panel shows all the files in the project and the center panel the current open file. At the bottom there is a console showing the status of the different tasks that Eclipse is performing. To the right is an outline of the methods and variables in the current file. This is generated automatically and helps the user easily overview and navigate the code.

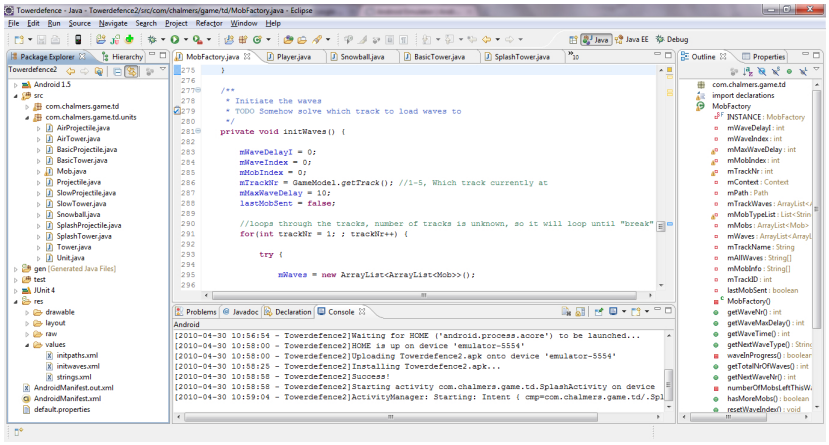

Figure 3.2.: The eclipse Intergrated Development Environment

The Android Development Kit includes an Android emulator. 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. As shown in figure 3.3, there is also access to the buttons normally available on an Android phone. The emulator can be set up to use different versions of Android and different resolutions, allowing the developer to verify compatibility. CITE HERE ””

Figure 3.3.: The Android emulator

4. The resulting game: Eskimo Tower Defense

4.1. Structure

At the very beginning of the development stage 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 on the game field. This worked well while experimenting with Android graphics. However, when menu development started a few weeks later, it was clear that more activities would be needed. A set of activities were created, everyone of them representing a different screen in the game.

The first activity being created is called SplashActivity. This is used only to display a splash screen for a few seconds. When the splash screen has been shown for those few seconds a new activity called Menu is created. The SplashActivity is then finished and removed from the activity stack. The Menu activity is not finished every time a menu item is clicked. Instead it remains on the activity stack, waiting to be shown after overlying activities are closed.

Four different activities can be started from the main menu: MenuGame, MenuHelp, MenuOptions and MenuCredits. These activities, together with the activity Menu, are the foundation of the game. The following chart visualizes the flow between different activities:

Arrows in the flowchart above (figure 4.1) 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 route map. Originally, only the game field View was shown in MenuGame. The progression route map was implemented at a later stage of development, and the idea of changing the View of MenuGame seemed like the most simple solution. With this solution the transition from the progression route map to the game field is performed by the different Views themselves (e.g when a track is started from the progression route map, the game field View is set as the visible View in MenuGame).

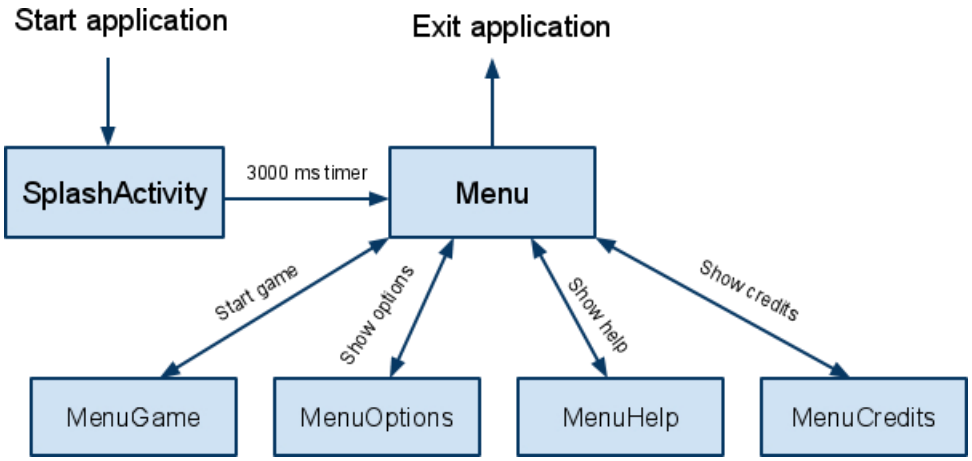

Figure 4.1.: Flowchart of activities in Eskimo Tower Defense

Not only Activity and View classes exist in the implementation of the game. When the View classes are created, a thread is also created for each View class. These threads continuously update the Views with regards to updating states and sound, and drawing whatever the View should display.

The classes MobFactory, Path and Highscore are used to translate XML-files containing information about different 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 menus and selecting a track on the progression map, the game is started. 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 objects' states in the game. It only contains a reference to GameModel which keeps track of all the data related to current states of the objects in the game. The GameView's task is to translate the data of the game into bitmaps drawn onto the screen and handle the user's input.

In the constructor, the GameView initializes the GameThread class. This class is a thread that runs during the game. It constantly invokes the updateModel(), updateSounds() 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, e.g. a method to terminate the thread when exiting the game.

The code above (figure 4.2) is placed in the run() method of the GameThread. The while loop is run as long as mRunThread is true, which it is as long as the game is running. The three method calls updateModel(), updateSounds() and onDraw() are

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

Figure 4.2.: Caption for GameThread

placed inside a synchronized statement. This blocks the rest of the program so there is no interference from the user input or other threads while a frame of the game is calculated and drawn to the screen.

The function of updateModel() is to update the model. The model is represented by the class GameModel that contains a list of all the objects on the screen, the player statistics, the path and some other properties of the game. The updateModel() method does all the calculation 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().

4.2. 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 are also utilized to some extent, but users are not required to use the physical buttons to play the game. An 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.

4.2.1. 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 pressing, 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: STATE_RUNNING, STATE_PAUSED, STATE_GAMEOVER and STATE_WIN.

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

Figure 4.3.: Caption for in-game pause menu codesnippet...

The statement above compares the coordinate 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 STATE_PAUSED.

If a track is lost, the GameView shows an in-game menu with the options "Restart", "Go to map" and "Exit". If the user choses "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 STATE_GAMEOVER. Then it checks what type of event that was invoked: the event is 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.

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() method. When for instance the state is STATE_PAUSED, the updateModel() stops updating the game objects and the onDraw() draws the in-game pause menu.

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

Figure 4.4.: Caption

4.2.2. Physical buttons

Every Android phone has some physical buttons. As a standard most phones have physical buttons for raising the volume, lowering the volume, going back, opening a menu and turning the device on and off (see figure 4.5). Some of the phones 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 (see figure 4.6). It is possible to assign other actions to the buttons than what is normally intended. For example the volume down button could be assigned to pause the game.

Figure 4.5.: 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 4.6.: Caption

In this game both the menu and the back button pause the game, bringing the pause menu to the front by changing the GAME_STATE to STATE_PAUSED. Volume up and down adjust the volume of the in-game sound. Since different phones 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.

4.2.3. 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 following pseudo code 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 4.7.: 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 the type given as an argument. 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 4.7 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 sensor, int accuracy) {}
    public void onSensorChanged(SensorEvent event) {
        mLatestSensorEvent = event;
    }
};
```

Figure 4.8.: Caption

The SensorEventListener interface has two methods that must be implemented: onAccuracyChanged() and onSensorChanged(). According to the Android API CITE"" (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 sensorś values are changed, which happens every time the acceleration on the phone is changed. To avoid being flooded by events generated from this sensor, the caught events are always saved. Only the last event is used when updating the game model. If game objects would be instantly updated when catching the events, performance would become an issue due to the amount of generated events.

4.3. 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.

4.3.1. 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 activity 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 the buttons have been positioned using the XML editor but the customization of the buttons has been drawn using Adobe Photoshop. As shown in figure 4.9, the main menu contains a Relative Layout which contains a Table Layout with five buttons: StartGame, Help, Options, Credits and Exit.

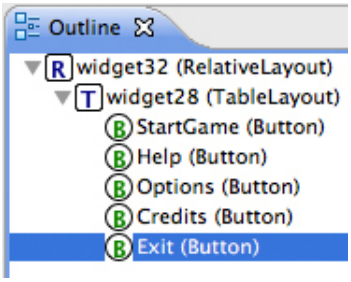

Figure 4.9.: Outline view of the layout editor in eclipse

Different properties can be set for the elements of the layout. As an 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 and releases the button.

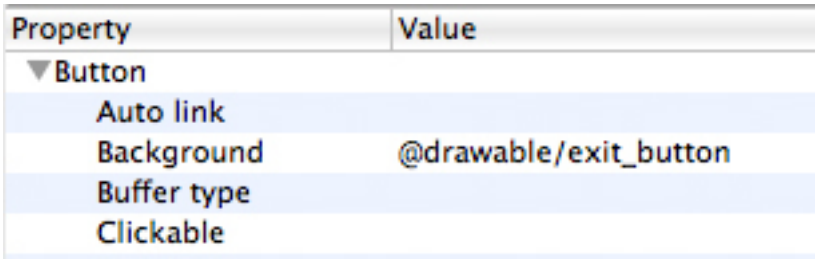

Figure 4.10.: Property window of the layout editor in Eclipse.

The XML-file (exit_button.xml), which is referenced from the Exit button's Background property in the main menu, contains the code shown in figure 4.11.

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

Figure 4.11.: Caption for exit button

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

Figure 4.12.: The exit button. To the left, not pressed. To the right, pressed.

4.3.2. 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 is being used in a cache, to prevent performance issues occurring due to excessive access to the resources of the application. The cache is implemented with a HashMap using the generated resource identifiers as keys. This ensures images are only loaded once.

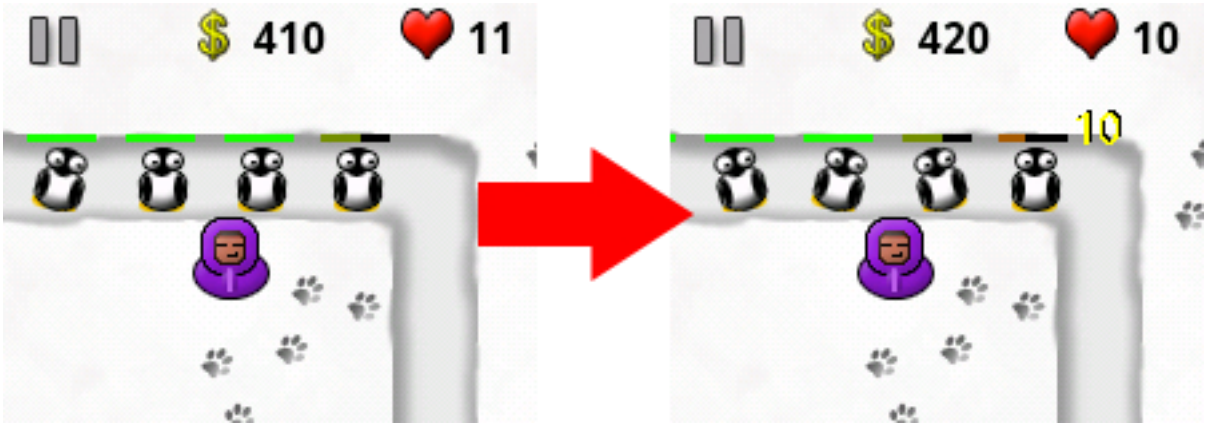

Figure 4.13.: Screenshot of a mob dying.

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 which brings the game to life.

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 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 4.14 .

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

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

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

Figure 4.14.: Example of how to draw graphic to the screen

4.3.3. Animation

Animating in Canvas is done by switching bitmaps with regular intervals. One of the animations is the water splash animation at the end of the map. When a mob reaches the final checkpoint it looks like 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.

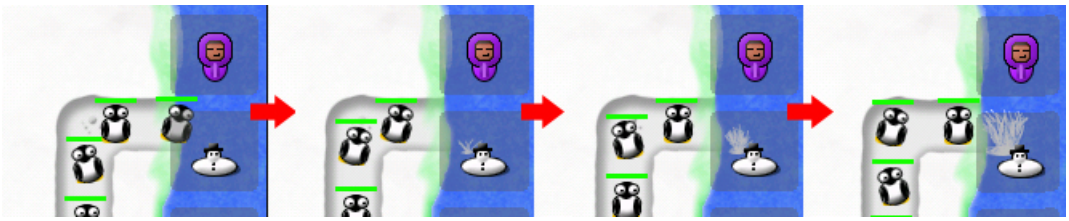

Figure 4.15.: Screenshot of water splash animation

The code in figure 4.16 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 what image will be drawn to the screen. Drawing different images in different intervals of this variable creates the effect of an animation. 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 4.16.: Caption for draw water splash..

4.3.4. Menus

Apart from the main menu that is defined in an XML-file, in-game menus are also implemented and updated in the method onDraw(). 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.

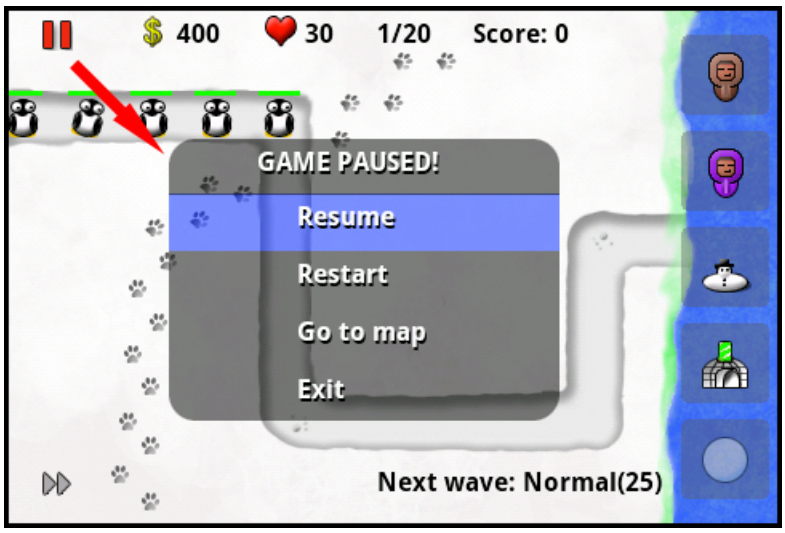

Figure 4.17.: Screenshot of in-game pause menu.

The menu shown in figure 4.17 contains five transparent images with some text on top of them. The images behind "Resume", "Restart", "Go to map" and "Exit" also exist with a blue background. These blue images are used when the user keeps his finger over the button, highlighting it. The user can also drag or swipe his finger over the buttons. The buttons are highlightened depending on the current position of the user input. It is only when the finger is released from the screen that a button is logically pressed. Handling user input this way prevents faulty user input, increasing the usability of the game.

4.3.5. Tower placement

Placing 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 but the player cannot put towers wherever he wants. The game field is divided into a grid of squares 16x16 pixels. There is no grid painted on the screen. Instead of showing the grid visually, the towers will snap into the positions on the game

field where they can be placed. It is not possible to build towers on the path, already existing towers, the pause button or 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 buttons is pressed and held down, a tooltip with information about that tower is dis played. As soon as the tower is dragged to the game field, the tooltip is removed and a image of the tower is shown. The tower image is not placed directly under the finger of the user, since it would be harder to see the tower during such circumstances. Instead the image is placed some pixels left of the user's finger. A screenshot of this feature is shown below in figure 4.18.

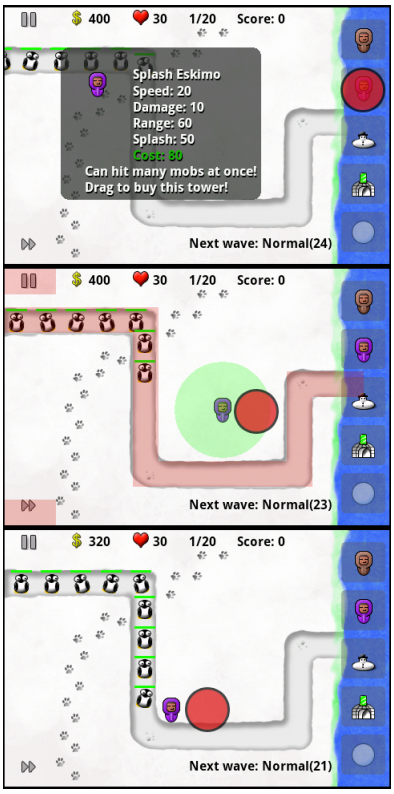

Figure 4.18.: Screenshots of a tower being built. The red circle shows where the screen is touched.

4.3.6. Tower upgrade

Upgrading a tower requires the user to press the tower on the game field. A tooltip will then be displayed with information about the towerś current level and next level attributes. 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 not be paused. This is not one of the four states that was described earlier. It is a smaller state in STATE_RUNNING which allows the toolbox to be displayed and the game still 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.

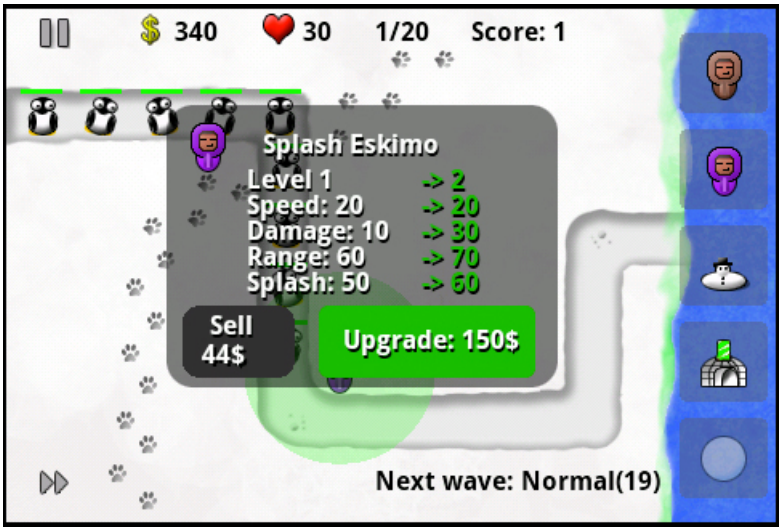

Figure 4.19.: Screenshot of the upgrade window

There are four different types of towers and they all have four different levels of power. The image of the towers reflect this (see figure 4.20).

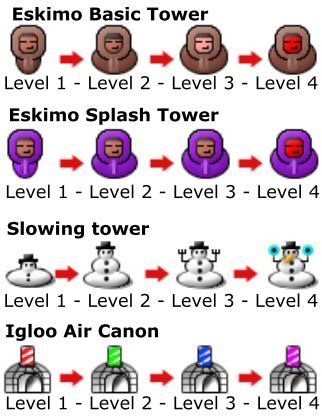

Figure 4.20.: All towers of the Eskimo Tower Defense

4.3.7. Mobs

There are five different mob types: Penguin, Bear, Polar Bear, Walrus and Flying Penguin (see Figure 4.21). Some of the mobs are animated. This is achieved in the same way as the water splash. 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.

Figure 4.21.: The different mobs of the game

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. The health-rectangle's length and color is calculated by the health of the mob. The less health a mob has the 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 following formula:

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

Figure 4.22.: Caption

4.4. Units

There are four different types of units in the game: mobs, towers, snowballs and projectiles. The mobs are walking down the path towards the ocean. The towers try to shoot the mobs to prevent them from reaching the ocean. The projectiles that the towers shoot are moving towards mobs, inflicting damage as they hit their target. The snowball is a special feature used for killing mobs. All these classes extend the class Unit. This class contains the methods and variables that are used by all these objects. A Unit object contains three instance variables that are set upon creation; a height, a width and a coordinate.

Coordinate is a custom class created to store and manipulate the coordinates of a unit. It only keeps track of two instance variables; x-position and y-position. To make handling of coordinates easier, there is a method called getAngle() that returns the angle between two coordinates that are given as parameters.

The method getDistance() returns the distance between two coordinates. The algorithm is the Pythagorean theorem and is implemented as follows:

```
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 4.23.: Caption

4.4.1. 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 super class.

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 loops through all mobs on the map and checks which mobs are within shooting range. Mobs that have walked a longer distance on the path are prioritized higher when the tower chooses its target. The method then returns a projectile corresponding to the tower type, by invoking the createProjectile() method.

Upgrading the towers is done by invoking the method upgrade(). Doing so alters the values of different attributes of the tower. The upgrade() method increases the variable keeping track of the level of the tower by one. It then uses this variable to fetch the new attributes from an array of predefined values, using the level number as index.

4.4.2. Mobs

Every mob is represented by an object of the Mob class. The implementation of mobs differ from that of towers in that there is only one class, instead of subclasses for each type. The mob type is set in the constructor, which is only invoked by MobFactory. The most important attributes of the Mob class follows:

- **mMaxHealth:** The maximum health of the mob
- **mHealth:** The current health of the mob. Never bigger then mMaxHealth
- **mAngle:** The current angle towards the next checkpoint
- **mSpeed:** The speed the mob moves over the map. Different speed for different types
- **mRevard:** The mob type (NORMAL, AIR, FAST, IMMUNE or HEALTHY)
- **mDistanceWalked:** Keeps track of how far the mob has walked. This is used to calculate highscore
- **mobImage:** A reference for the image to be drawn on the map
- **mPath:** The path the mob follows across the map
- **mCheckpoint:** The index of the current checkpoint the mob is walking towards

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 first updates the angle of the mob, making it point towards the next checkpoint of the path. After doing so, it updates the mob's position.

updatePosition() 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 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 route map. A new object of the class GameView is instantiated, and the method startTrack() is invoked. In method startTrack() the only instance of MobFactory is invoked with the method getInstance(). When initiated the context has to be sent to MobFactory, using the MobFactory method setContext(). This is needed to get the reference to the applications context, to be able to reach the resources that contains the XML-file initwaves.

When the method setContext() is called, the private method initWaves() is invoked. This method reads the file initwaves, which is structured as in figure 4.24

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

Figure 4.24.: 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 the 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 (”track_1”, ”track_2”, . . . , ”track_n”), and 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 to determine which type the mob should be, how many to create and how much health each mob should have. These strings are handled as shown in the code snippet below (figure 4.25).

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

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

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

Figure 4.25.: Caption

When the String has been processed, a third loop creates the given number of mobs and stores them in an ArrayList. When all mobs are created the list of mobs is stored in another ArrayList to distinguish waves that belong to different tracks.

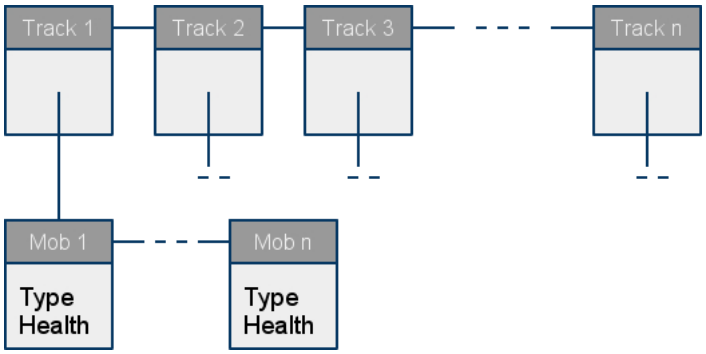

Figure 4.26.: Data structure of a wave object

MobFactory provides a method called getNextMob() that returns a Mob object. This method knows which track the player is currently playing, with help of the GameModel, and also knows which wave to send mobs from.

Since the 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.

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

Figure 4.27.: Caption

4.4.3. 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
- **mDamage:** The 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 a 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.

4.4.4. 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 4.28 is a very simplified example of how snowball objects handle accelerometer events.

```
public void updatePosition(SensorEvent s) {

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

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

Figure 4.28.: Caption

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.

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.

4.5. Path

The class Path represents the road that the mobs walk along. Each track has a unique path. Path has a list of the coordinates for the 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 two classes. Firstly, the mobs use it when they update their position on the map. They use the coordinates to calcuate in which direction they should move. Secondly, GameView uses 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, meaning that only one object of the class can exist. When the track is changed, the Path object is 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, which is structured as a string array, where every item is a checkpoint. Figure 4.29 shows the structure of initpath.

The initiation of Path is structured in a similar way as MobFactory. It contains two nested loops to iterate over every $< array >$ tag which represents a track and every $< item >$ tag which contains coordinates for the checkpoints. The data is also saved in an ArrayList, where every element contains an ArrayList of coordinates.

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

Figure 4.29.: Caption

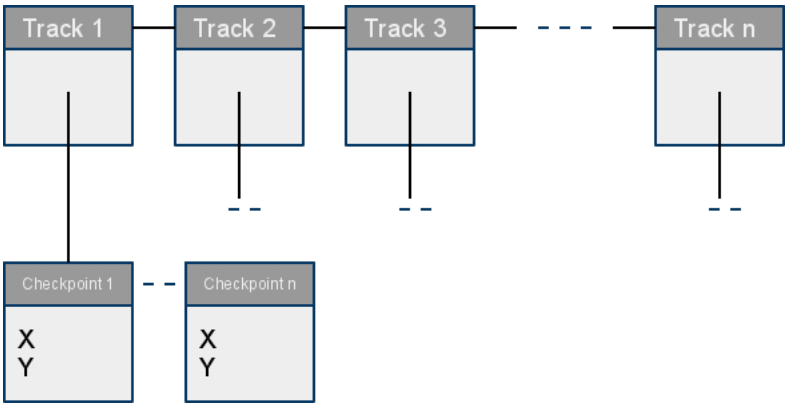

Figure 4.30.: Caption

4.6. 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 mob, not only does he loose a life but he also misses the reward money. This makes it harder to defend against the increasingly healthier mobs.

The points are added to the playerś total score for the current track. The score is calculated based on the starting health of the killed mobs. It is also decreased by a factor between 1-2 depending of how far the mob has travelled before it is killed. This introduces some variation to the high score, ensuring players are rewarded for a more efficient defense. With the distance included in the formula the highscore get bigger if a player kills the mobs early.

The high score is managed by the singleton named Highscore. When first initiated, it tries to read from the file tddata.txt. If the game has started for the first time, or if no track have been completed, the file does not exist. An exception that states that no such file exists, is thrown and caught. When caught, a file is created with the name tddata.txt.

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

Figure 4.31.: Caption

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

Figure 4.32.: Caption

Since the read and write methods are used on more than one occasion, the code for reading and writing has their own separate methods for initializing the BufferedReader and BufferedWriter, to reduce the total amount of code.

For the player to be able to progress in the game, a track has to be completed. When a track is completed the method saveCurrentTrackScore() is invoked and saves the score to the file tddata.txt. In the progression route map, the high score file is then read to determine if a certain track has been played. If a track has score stored the next track is then unlocked and becomes playable.

5. Discussion

5.1. Design choices

5.1.1. Fixed path versus maze building

The first important design decision that had to be taken was to decide 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 or maze building. With a fixed path the mobs move along a pre-defined route on the track and towers can be built along the sides. Maze building games on the other hand, consist of an open field where the mobs can move freely. It is up to the player to build obstacles forcing them to take detours towards 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) for Android which featured maze-style gameplay. Robo Defence is one of the most popular Tower Defense games currently on the app-market, distinguishing the game from from this popular alternative was one big factor in deciding upon the fixed path design.

There are many benefits with the 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 get 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, 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 deciding the theme of the game. Since no one in the team had any graphical experience 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 occuring. 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. In addition to the standard units of a Tower Defense one new special unit is implemented; the snowball. The snowball is a powerful unit which rolls around the screen to run over the mobs as the player tilts the phone.

5.1.3. Projectile algorithm

5.1.4. Waves

A big issue was how the waves were supposed to work. In other Tower Defense games like Bloons and Element Tower Defense the game pauses between every wave, giving time for the user to build more towers. Another way is to make the waves come continuously but with a delay between them. It was decided to use the delay approach. This to ensure that the game flows better and to remove unnecessary actions. Since the tracks feature lot of waves this might be irritating 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 alot more health than a normal mob taking a more considerable effort to kill. Since there only is one mob on boss waves, the countdown until the next wave starts immediately after it is created. To solve this we simply increased the delay after slow boss waves.

5.2. Concept

5.2.1. 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 phones accelerometer. One of the ideas was to control the towers by tilting the phone. Another idea was that the player should be able to control the range of the towers. If the player tilts the phone to the left, the towers range will increase in that direction. Another idea was to control the speed of the mobs. When tilting to the left, it would create a slop to the left making the mobs moving left go faster and mobs moving right slower.

Studies did not find any other games that has multiple paths for the mobs so this would be an easy feature to implement that would make it more unique. Another thought was letting the user control which way the mobs go 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 was actually implemented was that of a snowball. The snowball will roll over the map controlled by tilting the phone, almost like the classical board game Labyrinth. 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 more balanced compared to normal mobs.

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

5.2.2. User interface

User interface is one of the most important areas when working with touchscreen mobile phones or small touch-screens in general. The small screen needs to hold a lot of infor-

mation. But, since the touch-screen is operated with the userś fingers, and some people might have bigger fingers than others, it is important that items are not too small. This can make the player irritated when trying to hit the buttons. You do not want to make them too big either which can result in leaving a cluttered interface.

You want to fit as much as you can 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.

5.2.3. Animations

Animations was 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 makes them look like they are wobbling back and forth when they are 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.

5.3. Game balance

For a game to be interesting and thus sellable, it must provide just enough challange 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.

5.3.1. Towers

Balancing of the towers was done to make sure that no tower was superior to the others. 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. Different maps contains different combinations of mob types means the user has to change his strategy to meet the challenges he faces.

5.3.2. Mob waves

The biggest part was balancing of mob waves. Each mob killed awards the player with money. To make the game balanced, the income should 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.

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

5.3.3. Snowball

The player get one snowball for each 4000 points that is collected. The reason for this amount is that a game normally results in a total of around 10 000 to 15 000 points. This is done to limit the amount of times the user has access to the snowball, and two to three times per map recieved good feedback in testing.

The damage of the snowball was also up for discussion. At first, the snowball killed the mobs instantly. This was later changed to deal damage equal to 8% of the current health of the mob, each frame. The snowball was also modified to do less damage for each frame it spent on bosses. This because they have around 15 times more health than normal mobs. If the snowball would deal the same amount of damange to the bosses, the snowball would be too powerful.

5.4. Insight

Text...

5.5. 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 are discussed.

5.5.1. Public highscore

To make the game more attractive and addictive, which was one of the purposes 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 others by trying to beat their highscores. A server was needed to be able to upload and store the users scores. This was considered to take too much time and was not that important for the total game experience, therefore not implemented.

5.5.2. Fixed frame rate

If the game was ever to have a public highscore, the game need to be fair. If the game is run faster on phones with faster processors it would not be fair. Then the players with slower phones would have an advantage towards the rest of the players. To solve this the game has to run in the same speed independent of the speed of the phones processor. There is also a problem with future phones with superior processor speed. 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 achieve this effect. These were excluded from this project in favor of adding features that more directly affect the game experience. This issue is one of the highest prioritised of future work.

5.5.3. Sounds

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

5.5.4. Achievments

Another thing that might increase the game's lifespan is achievements. This is a form of extra bonuses given to the player for completing 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 against after completing all the standard maps in the game.

5.5.5. Snowball improvement

The snowball is one of the main parts that separates us from other Tower Defence 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 the player might not use it. He would then miss out of one of the things that makes the game unique. Further development of the snowball is therefore a important part to focus on.

Since the snowball is never introduced properly, new players might not notice 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 indicate 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. Now the snowball is able to roll over the water. One suggestion was that is should be sink or melt faster in the water. Instead of killing the mobs, there was an idea to have the snowball stun the mobs. This would stop them from walking for a moment so the towers had more time to shoot them.

5.5.6. More bonus weapons

One of the unique parts of our game is the snowball which is controlled by the tilt of the phone. An addition to the game could be to extend this idea to more player controllable 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 snowball falling down from the sky was also discussed. All these ideas would further make our game more unique and stand out from competition on Android Market.

6. Conclusions

7. Future work

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

7.1. Fixed time step

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

A. Appendix

List of Figures