System Design Document for Tank Revolution, an Android game application written in Java using the framework libGDX

Version: 2.0

Date: 2017-05-25

Author: Jakob Erlandsson, Anton Hägermalm, Simon Kärrman, Jakob Wall

This version overrides all previous versions

Contents

1 Introduction

This document describes the game application Tank Revolution and its construction. Tank Revolution is a turn based artillery game in 2D where there it is two to four tanks playing against each other. Every tank can be controlled by either an user or the computer. The objective is to be the last tank standing and to reach that goal the player can move its tank and then fire projectiles towards the other tanks. The ways to destroy tanks are ether by reducing health with direct hits and or destroying the ground under the tank so that the tank fall out of the screen.

1.1 Design Goals

The goals when making this application was to make it easy to expand and add functionality. It should also be loosely coupled so that the model is independent from the GUI and Controllers. The application should be testable and therefore the model also needs to be decoupled from the framework.

1.2 Definitions, acronyms and abbreviations

- **Player** - an actual player of the game.
- **NPC** - (Non-Player Character) players in the game controlled by the application.
- **libGDX** - a platform independent framework for making games.
- **STAN** - a dependency analysis tool
- **Pull based MVC** - a kind of MVC where the view pulls the information it needs from the model to be able to draw what it needs to draw on the screen.

2 System Architecture

This application uses a pull based MVC-architecture. The application is intended for the platform Android with the target SDK: Android 25 (Nougat).

2.1 Main Development Language

Since *Tank Revolution* is developed for Android devices the chosen development language was Java. Most Android applications use Java since it's the native language of the operating system[1] even though it is possible to develop in other languages.

2.2 Game Engine

The application is built based on the game/physics engine libGDX, developed by Bad Logic Games in 2014[2]. This helps by having a lot of the difficult and time consuming code already written so that it can be used by any application that imports it. The physics part of the game engine is handled by a plug-in called "Box2D", a physics engine written in C++ by Erin Catto in 2007.

2.3 libGDX

A libGDX application is structured with a main class that implements *ApplicationListener*. The most important methods is the *create* and *render* methods. The *create* method is called when the application is launched and have the responsibility to initialize everything that is necessary for the application to run. What makes this a game engine is the *render* method. This method is called 60 times per seconds and it is here the graphical rendering is supposed to happen.

2.4 Terrain Handler

One of the key features of the game is that the terrain should be destroyable. When a projectile hits the terrain it should make a hole in the terrain as big as the blast radius of the projectile. To make this possible a library called libGDX-Box2D-Destructible-Terrain by Quails Hill Studio was used [3]. This library was in some cases too specifically designed for the project that it was first made for. Therefore some classes had to be rewritten and some moderators created to make it possible for the environment classes in the project model to use the functionality of the library. The core functionality that was used from this library was its algorithms for polygon clipping.

2.5 Garbage Collector

Some classes in both libGDX and Box2D is not affected by the Java Garbage Collector, Java GC. In the case of Box2D, it is written in C++ which have no equivalent to the Java GC and therefore objects have to be disposed manually. The graphical classes in the libGDX library such as those handling images, sounds and fonts are managed native drivers[4] and are therefore not affected by the Java GC. This objects also need to be disposed when not used any more otherwise it will take up a lot of RAM.

3 Subsystem decomposition

This is the package structure of this project:

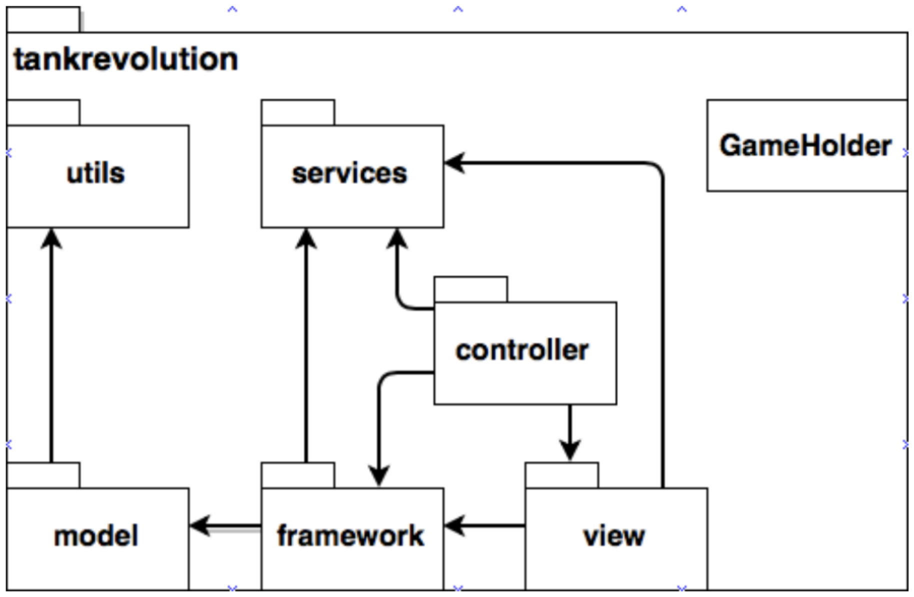

Figure 1: Package UML Diagram

3.1 Separation from framework

In order to make the application more open to future expansion, we wanted to completely separate our MVC from the framework provided by libGDX. To do this all the framework specific code was moved to a class called *Environment* in the package *framework*. In front of this class an interface *IEnvironment* was added as a facade in front of the framework specific code. The *IEnvironment* interface also works as a wrapper of the model. All the events from the controllers and all the pulling of data from the view goes through the *IEnvironment* interface. The result of this is that the logical model is completely separated from the framework.

3.2 The responsibility of *GameHolder*

Since this is an libGDX-based application it's necessary to have a main class that implements *ApplicationListener*. This class is *GameHolder* in this project and since this is the class that is called by the frameworks gameloop this have the responsibility to handle which controllers that should be active and which view rendered.

3.3 Pull based MVC

libGDX uses a pull based architecture when it comes to graphical rendering. This because it is continuously rendering[5] and not just rendering on command. Because of this to be able to have a consistent data flow it was decided to use a pull based MVC structure throughout the whole project. The natural choice when using the MVC pattern is to have a push based structure. Initially a push based structure was used. But since libGDX contradicts that structure it was impossible to make the project completely push based. Therefore the decision was made to make the whole structure pull based to have a consistent flow of data.

3.4 View

3.4.1 Viewable

The application contains different views. For example there is one view for the Start Menu, one for the game, one for custom setup and so on. All the different views implements an interface called *Viewable* that looks like this:

```
public interface Viewable {
   void update();
   void dispose();
}
```

This is a contract for classes that implements this that they will have graphical components to draw on the screen. In this way *GameHolder* don't have to keep track of which view is rendered just that it's rendering it.

3.4.2 Abstraction of *GameView*

The *GameView* class handles the rendering of the graphics connected to the actual game. There is a lot of different components in that view and to make this class more abstract the different components is handled by different classes (see Appendices figure 4). For example there is a class called *GraphicalTank* that handles the drawing of a tank and another class, called *GraphicalTerrain*, for drawing the terrain.

3.4.3 The view-instance in *GameHolder*

Since the view is rendered 60 times per second it's necessary to update the environment at the same rate. Therefore the *GameView* calls the update method in *Environment* every time *GameHolder* calls the update method in *GameView*. This makes the game appear to pause itself automatically if the view-instance in *GameHolder* is changed. The game is resumed if an instance of *GameView* is rendered again.

3.5 Dependency Analysis

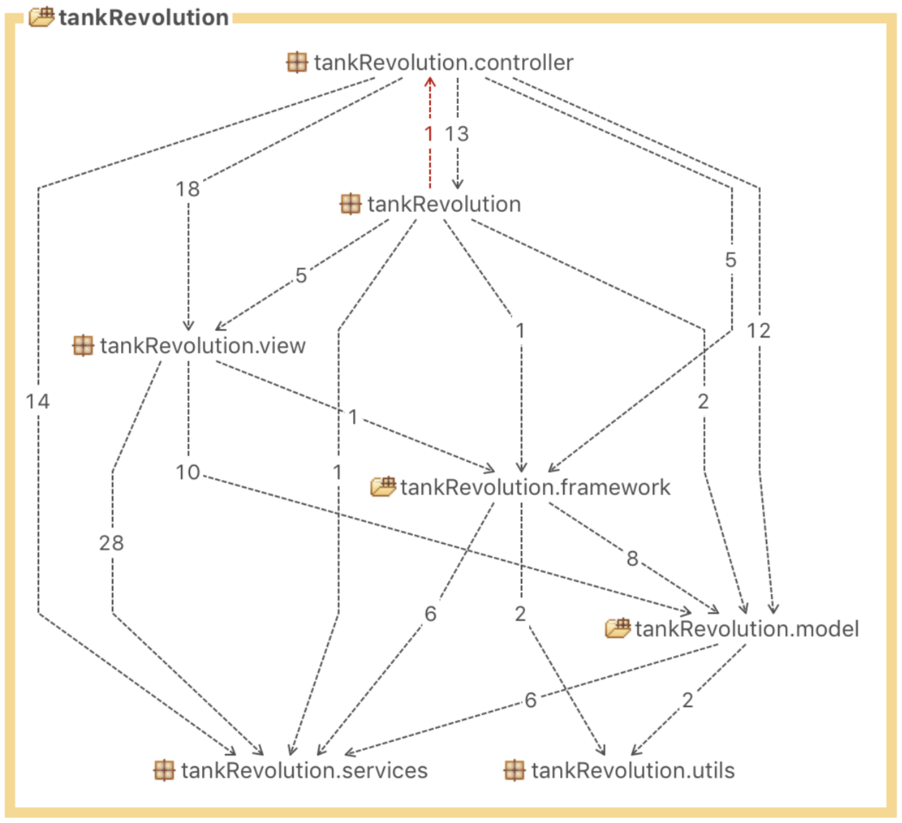

Figure 2: Dependency analysis diagram generated by STAN

As figure 2 displays there is one circular dependency. That is completely because of the structure of libGDX. The application has some different screens (a pair of one view and one controller) and it is the main class (*GameHolder*) that render this screens. To be able to change the screen when a user presses a button an interaction with the main class has to be made. Therefore the different controllers needs a way to communicate with the main class and since it is the main class that creates the controller the circular dependency is unavoidable.

3.6 Tests

The program has comparatively few tests. The reason for that lies in the use of libGDX. As mentioned before, much functionality in the program depends on the implemented framework. For example is the implementation of a projectile being shoot a call to libGDX's physics engine to add a projectile body to the world. How physically the body relates in the world to the environment is dealt with by the framework itself. We do not test the framework but trust is to work according to our expectations.

What we can and do test is that functions responding to actions of the framework is dealt with correctly. We test basic functionality of the tank, NPC and terrain, all things separated from the framework.

3.7 Sequence Diagrams

To describe two how two use cases are implemented in code there are sequence diagrams made to display the call chains. The use cases described are Fire, see appendices figure 7, and Switch weapon, see appendices figure 8.

4 Persistent data management

The project uses a lot of assets such as images, fonts and sounds. To handle all the assets there is a class called *AssetsManager* that standardize all the assets handling. This class have the responsibility to load assets and provide classes that needs them with the specific assets. To avoid that assets have to be loaded more than once the *AssetManager* uses the Singleton pattern. This way there is never any risk of an asset being loaded twice. The assets aren't all loaded at once, the first time the start menu is launched the assets that is needed for that is loaded. The assets for the game is loaded when the first game is started so that there will be the minimal amount of loading time.

5 Access control and security

NA. Application is launched and exited the same way as any android app.

6 Concurrency issues

NA. *TankRevolution* is a single threaded game.

References

[1] ”The Android Open Source Project on Open Hub: Languages Page”, Openhub.net, 2017. [Online]. Available: https://www.openhub.net/p/android/analyses/latest/languages_summary. [Accessed: 26- May- 2017].

[2] ”libGDX 1.0 released”, Badlogicgames.com, 2014. [Online]. Available: http://www.badlogicgames.com/wordpress/?p=3412. [Accessed: 25- May- 2017].

[3] O. Panczuk, ”Destructible terrain in libGDX - Quails Hill Studio Blog Article”, Quailshillstudio.com, 2016. [Online]. Available: http://www.quailshillstudio.com/en/blog/article/00001-destructible-terrain-libgdx-box2d-polygon-clipping-algorithm/. [Accessed: 25- May- 2017].

[4] ”Memory Management”, GitHub - libgdx/libgdx, 2017. [Online]. Available: https://github.com/libgdx/libgdx/wiki/Memory-management. [Accessed: 25- May- 2017].

[5] ”Continuous non continuous rendering”, GitHub - libgdx/libgdx, 2015. [Online]. Available: https://github.com/libgdx/libgdx/wiki/Continuous--non-continuous-rendering. [Accessed: 26- May- 2017].

Appendices

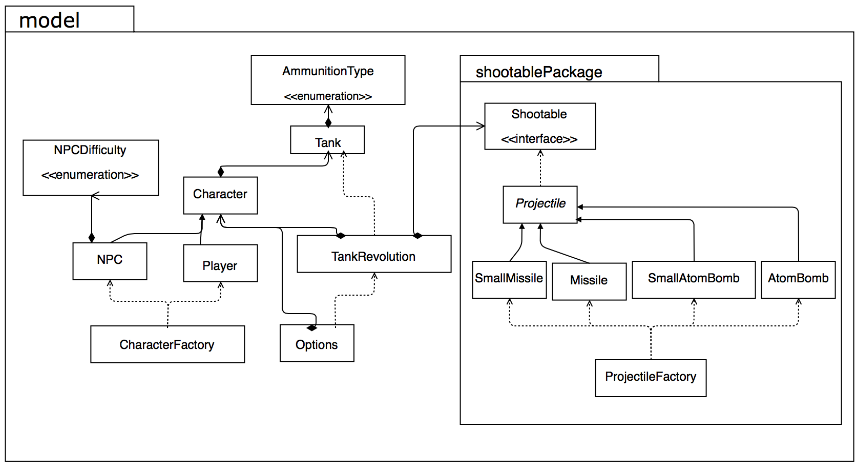

Figure 3: UML class-diagram of model package

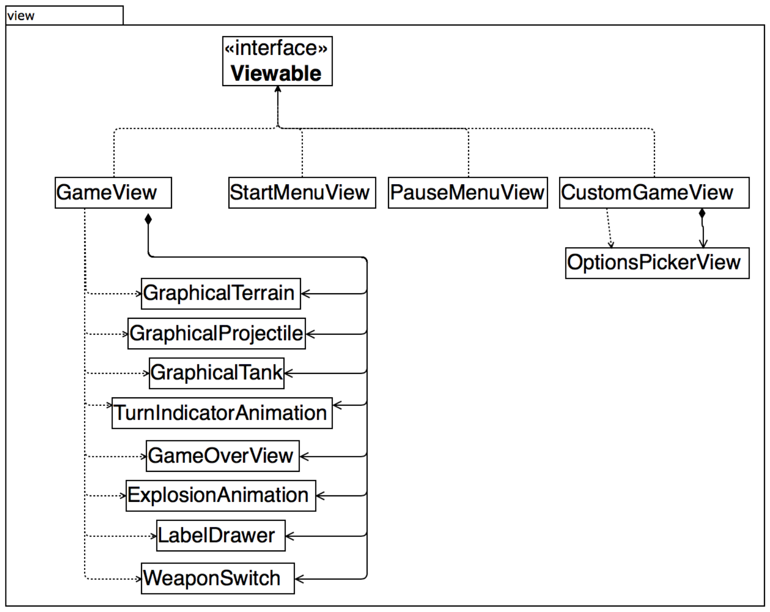

Figure 4: UML class-diagram of view package

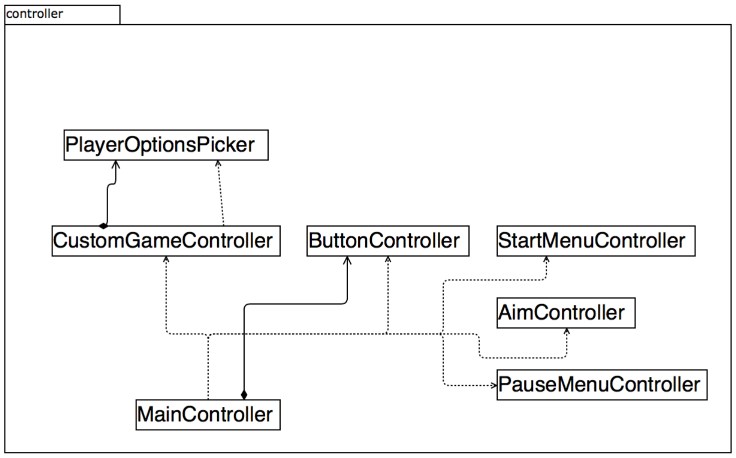

Figure 5: UML class-diagram of controller package

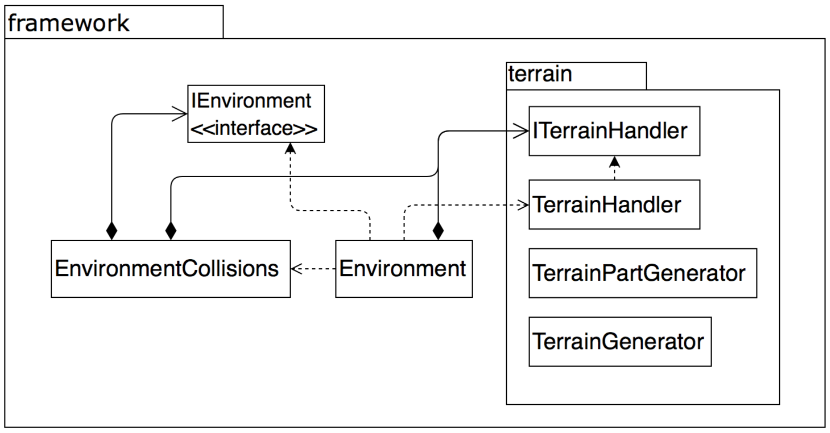

Figure 6: UML class-diagram of framework package

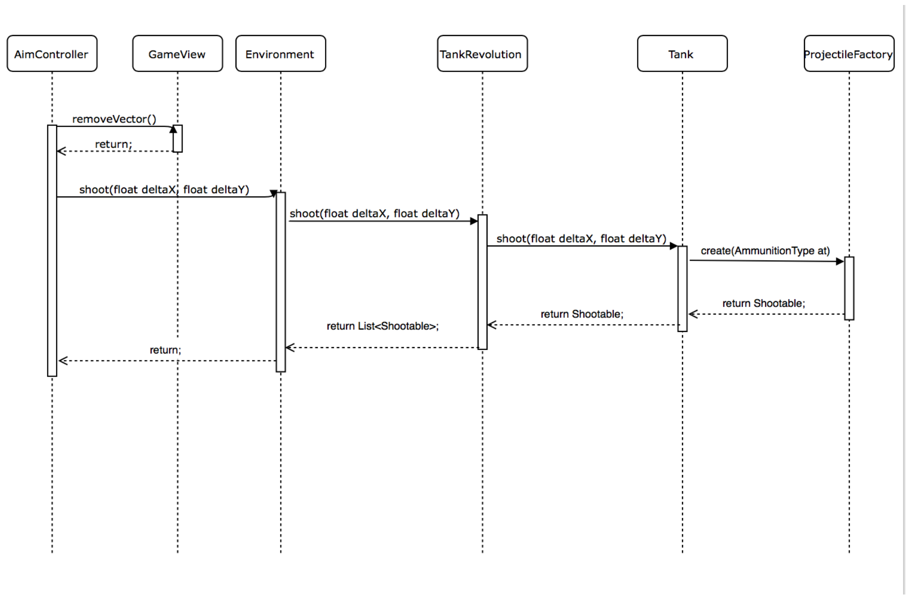

Figure 7: Sequence diagram of Use Case: Fire

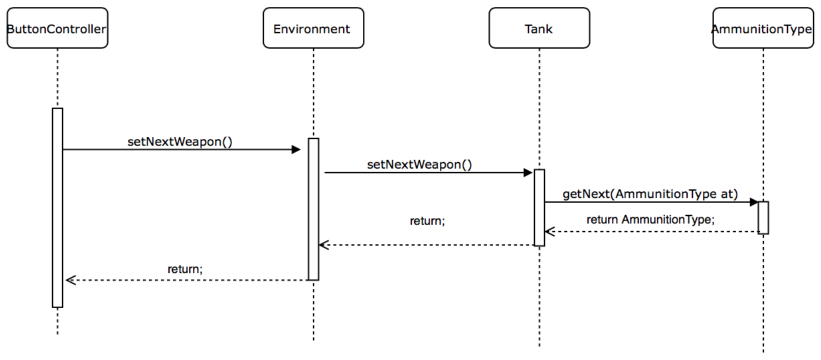

Figure 8: Sequence diagram of Use Case: Switch Weapon (the methods setNextWeapon() and getNext() is changed for setPreviousWeapon() and getPrevious() to get the previous weapon).