System Design Document
Project group 5 - Zombieweek

Neda Farhand
Tobias Hallberg
Daniel Johansson
Erik Öhrn

2015-05-31

Version and date:
0.1 2015-04-29
0.2 2015-05-03
0.3 2015-05-04
0.4 2015-05-05
0.5 2015-05-25
0.6 2015-05-26
0.7 2015-05-27
0.8 2015-05-30
0.9 2015-05-31
1.0 2015-05-31

Contents

1 Introduction

This is the system design document for the game ZombieWeek.

1.1 Design goals

The design of ZombieWeek aims to implement Model-View-Controller. That is, the different classes will be split up in different packages depending on what they do. In theory this means that coupling will be minimized and the game's structure easy to replicate if needed. The different libraries, such as libGDX and Box2D, should also be able to be switched out to libraries with similar properties and classes using adapter classes. This means that all classes that handles either Box2d or Libgdx classes and functions need to be separated from the rest of the program.

1.2 Definitions, acronyms and abbreviations

Due to the technical aspect of this project there will be terms that might be unknown to the reader. These are included below.

Avatar - The player's character, in this case either Emil or Emilia.

Java - A programming language able to create programs that are runnable on several different systems.

GUI - Graphical user interface. The part of the game that the player sees.

HUD - Heads-up display. A way to look at instruments and gauges without having to look down, in the case of video games this is often a layer on the game screen with information about the player's health, ammunition et cetera.

Path-finding - Finding the shortest route between two objects without colliding with anything. In video games this is often done to simulate artificial intelligence, moving for example an enemy closer to the player. The path finding in ZombieWeek uses an A*-algorithm

A* - An algorithm for finding the shortest path between two objects, similar to Dijkstra's algorithm but more efficient for game programming purposes.

libGDX - A Java library for constructing games. It has classes for rendering images, creating a game screen and other useful applications.

Box2D - A Java library for physic simulation.

Gradle - A program used for compiling and build automation of software projects. Similar to Apache Maven, but doesn't use XML.

MVC / Model-View-Controller - A design pattern where the classes are divided into three categories: model, view and controller. These store, modify and present the data, respectively.

Body - A representation of a physical body in the game world.

Step - Updating the physical game world.

World - A representation of a world, containing e.g. body data and gravity settings.

Sprite - 2D image

Book - The player's weapon. Can be thrown.

Tiled - A program for creating the tiled maps which are used in this game

Façade - A design pattern for separating libraries

STAN - A software structure analysis tool

2 System design

2.1 Overview

The application uses the MVC design pattern. The model contains data for e.g. the game world, player and enemies. The views render data from the model and shows it to the user as a GUI, this includes e.g. the menu screen and the game screen. The controllers will take care of user input and change the models data, turning user input to game actions.

2.1.1 Model

To avoid a large and cluttered model, the different objects have been divided into several classes. The main model, GameModel, holds a player, which in turn has a sprite, body etc., several rooms containing zombies and worlds, and so on.

2.1.2 Rooms

The room will contain the graphical representation of the physical world of each game room.

Each room will store zombies, potions and books as array lists so they easily can be fetched and iterated over. When a room changes these can quickly be loaded together with the map and world, so that the entities themselves won't disappear or respawn. The tiled map is a TMX file created with Tiled. It stores the image layers as well as a layer containing metadata for each tile.

In order to create all the objects in the room all tiles in its corresponding tiled map will be checked for different predefined properties. If a certain property is found the objects spawn controller will place the correct objects in the room. In this way all walls, doors, potions, spawn points etc. are created.

Every room belongs to a level, and all levels are stored in the GameModel.

Switching rooms It should be possible to change room at any given time, but especially if a player walks through a door. The model stores where the player will be placed in the new room and which room that shall be loaded.

The player will keep its data intact (excluding friction and a few other room dependent variables), and all zombies, books and potions are not to be removed in the old world.

Room update During each frame update, 60 times per second, the world updates as well (if the room isn't being changed). This means that the world will step, the entities move and the contact listener will listen to any objects colliding. All objects set for removal will be removed here, for example the player body, if the room has changed, or a book, if it's been picked up. If the room has been changed, a new player body will be placed at the position set by the room switching method and the renderer will be updated to the current room. If the room hasn't been visited before all physical objects will be created at their given positions.

2.1.3 Entities

All entities have a body and a graphical representation of itself, a sprite or an animator. The body controls the physics of the entity, containing the mass, friction, position, velocity and making it collidable. The sprite controls the graphical representation of the entity with a 2D image. If the entity should be animated, the entity uses an animator to loop through multiple 2D images. An entity can be e.g. the player or a zombie.

Player The player is an entity controlled by the person playing the game. There's only one player per game, and it's stored in the game model. The player contains data such as health, ammunition, speed and so forth. It can be controlled by the user using the input controller class, which in turn call upon the player controller class modifying these data.

Zombies There will several different zombie types in the game, each which their own damage, health, speed etc. The main difference between a zombie and the player will be how they move. Where the player is controlled by the user, the zombies will chase the player. This will be done done using a path finding algorithm which finds the shortest path between the zombie and the player without the zombie walking into walls etc.

Potions There will be different potions placed in the game which alter the player's behaviour. The most important one is the health potion, which restores the player's health. For easy access, the potions in each room will be stored as a list.

Books A book will either be created on the ground, which is usually the case when a book is set to spawn at a specific place, or created when thrown. If it's thrown the player speed and the book speed is taken into consideration,

and the projectile controller spawns the book moving in both the thrown direction and the player direction using vector addition. The book will stay in the air for a short period, not colliding with books on the ground or low obstacles, and then hit the ground. When it hits the ground it will slow down until it comes to a stop. It can, after hitting the ground, be picked up by the player again, but won't do any damage if it hits a zombie. If it hits a zombie while in the air the zombie will take damage and be knocked back. The book will then drop to the ground.

2.1.4 Contact

There are multiple entities and bodies that are collidable in each room. A contact listener in each room world will be set, and a contact controller handling the events forwarded by the listener. By using each fixtures category bits, which stores which kind of entity the fixture belongs to, the controller can tell if an action should be taken or if will just let it be. If, for example, a book hits a zombie it will tell the projectile controller to apply a hit to the zombie using the book. If a zombie hits a wall, however, nothing should happen.

When a zombie makes contact with a player it should attack her.

2.2 Software decomposition

2.2.1 General

The application uses three main systems. Box2D, libGDX and our implementation. Box2D takes care of the physical world. libGDX is continuously rendering data to graphical output. Our implementation takes care of user input, player and enemy movement, and the graphical representation of the game.

Our implementation is decomposed into the following modules:

- model, data for all objects
- view, graphical representation of the model
- controller, controllers for the model
- utils, contains utilities e.g. path finding algorithm and constants adapter, graphically representable objects
- ZombieWeek is the application entry class

2.2.2 Decomposition into subsystems

As stated in the goal section of this document the game implements Model-View-Controller. The model classes store the game's data, such as x and y coordinates of the player and which levels that are available. The controller package contains classes which modifies the data and perform calculations. This can be moving the zombies or collecting input from the player, which needs to be transformed into a path and movement (updated x and y coordinates) respectively. In the view package the game screens are created and updated. Furthermore there are several helper classes, such as an implementation of the A*-algorithm, which are stored in a utility package.

To separate classes with connection to the frameworks Libgdx and Box2d, all wrappers and façade classes are put in a adapter package. Doing this enhances loose coupling, making it easier to change framework for the physics and graphics. This makes the model, nor the controllers, having no knowledge about the frameworks.

2.2.3 Layering

Figur över de olika paketen, göra i paint kanske

2.2.4 Dependency analysis

The dependencies are as shown in the figure below, created with STAN.

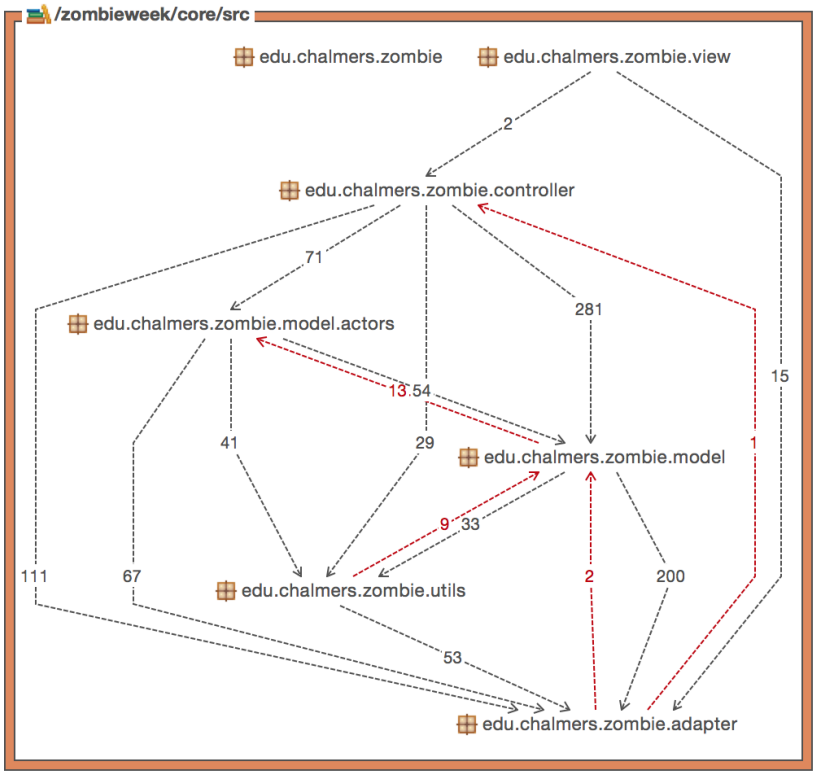

Figure 1: An UML diagram over the project

2.3 Concurrency issues

In case the player is to stop diagonally threads need to be used. This means that there is a slim possibility that the world will step at the same time the user stops the player. If this happens a run time error will be thrown and the game will crash. Therefore this feature is not implemented in the current version, but can be tested by setting a variable in the player class.

Overall, it's important not to transform any bodies during the world step, because of the way Box2d works. This is solved during the room switching by setting temporary variables in the model for what to happen the next time the room updates.

2.4 Persistent data management

Resources to the game are stored in a directory called assets. When the application is started it uses a resource manager to load the files. The resource manager has got support for images, sound and tiled maps using their wrapper classes. The classes can then get the resources from the manager,

through the game model. If the resource is no longer in need, it can be disposed.

In addition to this the application has the functionality to save and load games. At game start the application will search for a file called savedGames.properties. If the file is not found, it will create one. This is where the application stores data for e.g. the highest completed level. When player has completed a level, the application will automatically save the game and override the current data in the file.

2.5 Access control and security

Very few variables are public, and methods are public only when necessary.

2.6 Boundary conditions

N/A

A File formats

TMX A tiled map

JAVA A java source file

PNG An image with alpha layer