System design document for EGA

Version: 1.0

Date: 2015-05-14

Author: Emil Axelsson

This version overrides all previous versions.

1 Introduction

EGA is a one-player level-based, platform game. The users task is to get the character to the exitdoor. The exitdoor will always be closed, therefore the user will first have to collect the key. The character have two modes, small and big. The user have to switch between these two in order to get over the track. The user changes the characters mode by collecting big-/small stars. Big stars makes the character big, and small makes the charachter small. The game will also clock how long the user take on every level. The user can increase their best time by replaying the level.

1.1 Design goals

Our vision are to separate the controller-, model- and view classes. This pattern is called MVC (Model-View-Controller). If this pattern is followed, the application will be open for further implementations (Open closed principle).

1.2 Definitions, acronyms and abbreviations

- Java, the programming language that will be used.
- libGDX, API that will be used in the application (contains physics engine).
- GUI, graphical user interface. The things you see on the screen.
- Tiled, the program that will be used to create maps.
- Highscore, the best time on every level.
- The user - the real life person that plays the game
- Character - the center of the game, a player controlled object
- Level - contains a map and everything else visible to the user
- Map - the layout where the character moves
- Stars - an object the character picks up to grow or shrink
- Exit-door - the end of the map, this is where the user should aim to reach
- Spikes - dangerous objects, if touched resets the character to the start of the map
- MVC, a code design pattern where you split up the code for controll, the data and the view that will be shown.

2 System design

2.1 Overview

As written before, the application will be using the MVC pattern. Most of the controller classes will be using libGDX.

2.1.1 Observers

The entity models extends the Observable class. The view classes observers their respective model and act when something changes. For example, the controller class for the chracter changes the charachters position in the model when it moves and the view renders the character in its new position.

2.2 Software decomposition

2.2.1 General

- Main, the package that will start the application
- Model, the package that contains the data for the view. These classes observes the view classes when they should redraw things on the map.
- View, contains all the GUI modules (CharacterView and Menus for example).
- Controller, these classes will be handling the input from the user.
- Event, the classes that will be handling events.

Package diagram. For each package an UML class diagram in appendix

2.2.2 Decomposition into subsystems

2.2.3 Layering

See Figure 1 in appendix.

2.2.4 Dependency analysis

See Figure 2 in appendix.

2.3 Concurrency issues

The libGDX API is not designed to followed MVC. Therefore the implementation of this pattern is a problem. All the entities on the map will be needing 3 different classes, a controller, model and view. Which comlicate things more than necessary.

2.4 Persistent data management

The maps will be created in Tiled. Tiled save these maps in ".tmx"-formate. These files will be stored in a folder in the project. libGDX have a support for this kind of formate, and contains a method that will load the map.

The users highscore on every level aswell as the customized controls will be stored in a ".sav"-file. When the user restarts the game, the highscores will remain.

2.5 Access control and security
N/A

2.6 Boundary conditions
N/A

3 References

APPENDIX
Layering

Dependency analysis