Alesk The Unicorn

System design document

Version:

1.0

Date:

2018-05-25

Authors:

Kamil Miller
Johanna Torbjörnsson
Pontus Stjernström

This version overrides all previous versions.

1 Introduction

Alesk The Unicorn is a fast paced, whimsical 2D-platform game. The user plays the role as Alesk, a hungry unicorn searching for lunchboxes in the night. The goal is to eat all the food boxes on each level and not fall down or be killed by spiders, lava or dangerous spikes. On his way to saturation, different powerups can be used to help on the hunt.

The game is a focused experience, meant to be played in short sessions whenever there is time to kill. Therefore the different levels will be independent of each other and self contained units. The goal of each level is to minimize the time spent to collect all lunchboxes while trying to survive from the different dangers.

1.1 Definitions, acronyms and abbreviations

- Player character = Alesk The Unicorn itself
- Level = A standalone instance of the game, containing all game entities such as blocks, player, enemies etc. They do not need to be interconnected or coherent.
- JRE = Java Runtime Environment
- Application = the full application with menus and playable game
- Game = the playable part of the application

2 System Architecture

The application runs on a single desktop computer, without any connection to the internet. The environment required is the Java Runtime Environment, i.e. if installed, it can run on any OS supporting the JRE.

The application greatly uses the Java game library LibGDX for the technical parts.

2.1 Graphics, sprites and animations

Each drawable class has a getName() method in order to connect it to the correct sprite file. The sprite files are PNGs and can be found in the core/assets folder.

The player character is a special case, and have three different sprite files. Each sprite file represents a different pickup state, since the pickups can affect the appearance of the player character. Each player sprite file is structured as a sprite sheet, where each row represents a different moving state, e.g jumping, standing, running etc. Each column in the file represent a animation step. In the renderer, the correct sprite file and row is determined based the states of the player. The column is determined by the current frame number.

The enemies are also animated but has only one state, and thus has a single sprite file with a single row with several columns. The column is determined by the current frame number,

just like with the player character. The rest of the sprites are static, and therefore consists of only one image.

Some of the graphics is not representing anything in the model. Two examples of this are the game background and the rainbow, which are described below.

2.1.1 The background image of the game

The background is constructed by two different textures - one with clouds that are used in the bottom of the game screen and one with stars which is repeated indefinitely upwards. At all times, a maximum of two of each of these are rendered. Their positions are calculated in the view.

2.1.2 The rainbow

When the player character is jumping, falling or has picked up an energy drink, a rainbow is rendered behind it. The rainbow consists of a list of rainbow particles, and when rendered, each particle results in a rendition of a thin slice of the rainbow.

When the player character moves, the list of rainbow particles is updated. Particles are added at the position of the character if relevant. When the particles has existed for some time, they are faded and displaced, and eventually deleted from the list.

2.2 Collision logic

The collision detection uses standard Axis-Aligned Bounding Boxes (AABB) with an Intersection function checking whether we have two overlapping boxes.
If this is true we check for a proper response, which has 3 cases:

Player vs Pickup: Just trigger the pickup logic.

Player vs Enemy: Trigger taking damage logic on Player. If dumbell effect is active on player, trigger damage logic on enemy.

Character vs Platform: This is the hardest response, we check if the overlapping is coming from above, below, right-of, left-of. For each an exact re-correction on the character is done as to not get stuck inside the platform. For the left/right-of cases an additional look-ahead is made to see if there is a platform ahead of the character at the same elevation, in which case we do not re-correct from left/right but instead on-top. Without the look-ahead, if the space between platforms is sufficiently small, then the character will always get corrected as a right/left-of collision, therefore not allowing the character to run past gaps that should be possible.

2.3 Pickups

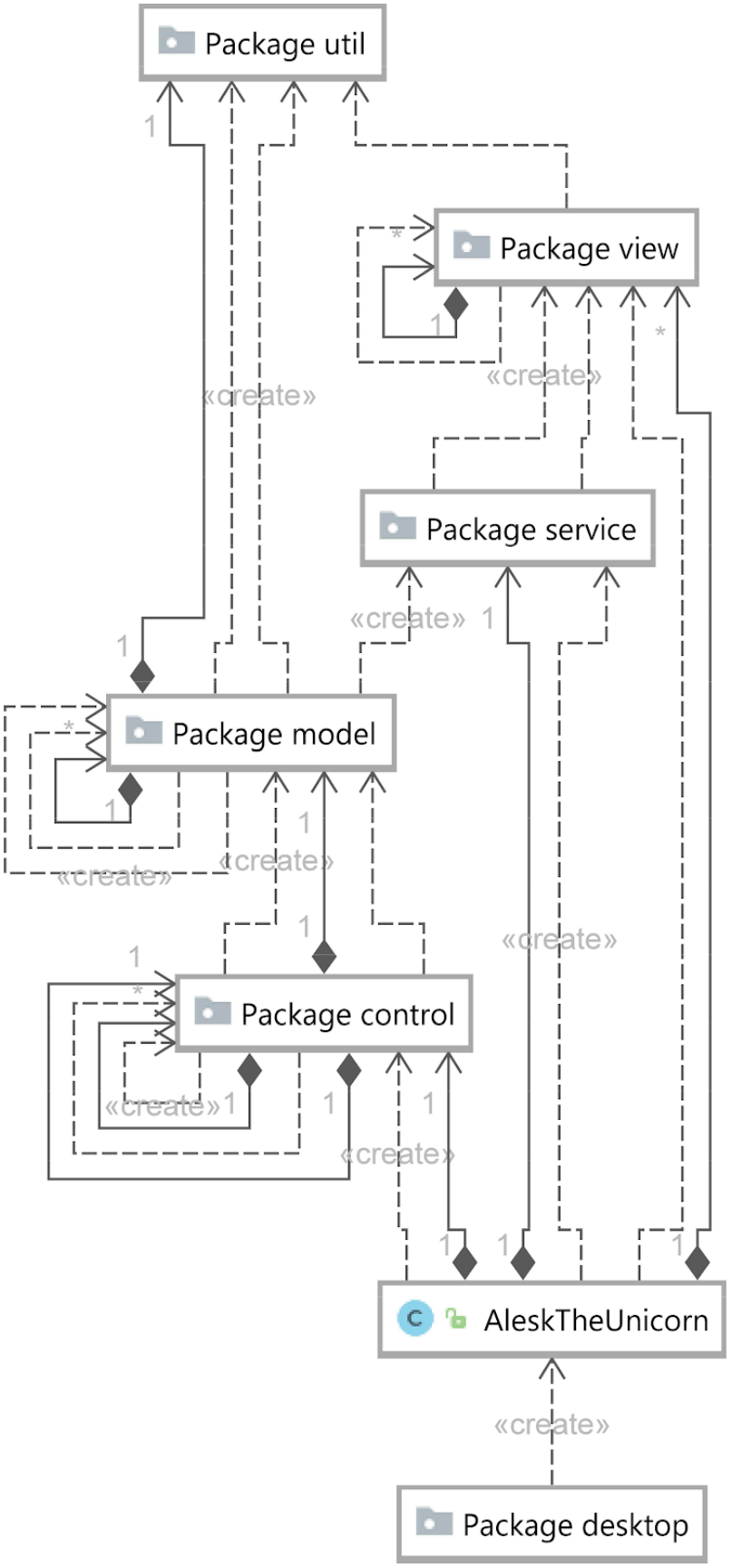
Pickups use either a player reference or a level reference which is used with its DoIt method. Each distinct pickup does something on that reference and schedules a deactivation of that

effect. Currently only **LunchBox** classifies as a level pickup, when triggered these tell the level that the number of picked up lunchboxes has increased.
For pickups that act upon the player, 3 are currently in the game: **Wings, Dumbbell, EnergyDrink**. Each of these change the state of the player somehow and schedule a TimerTask to revert this state change.

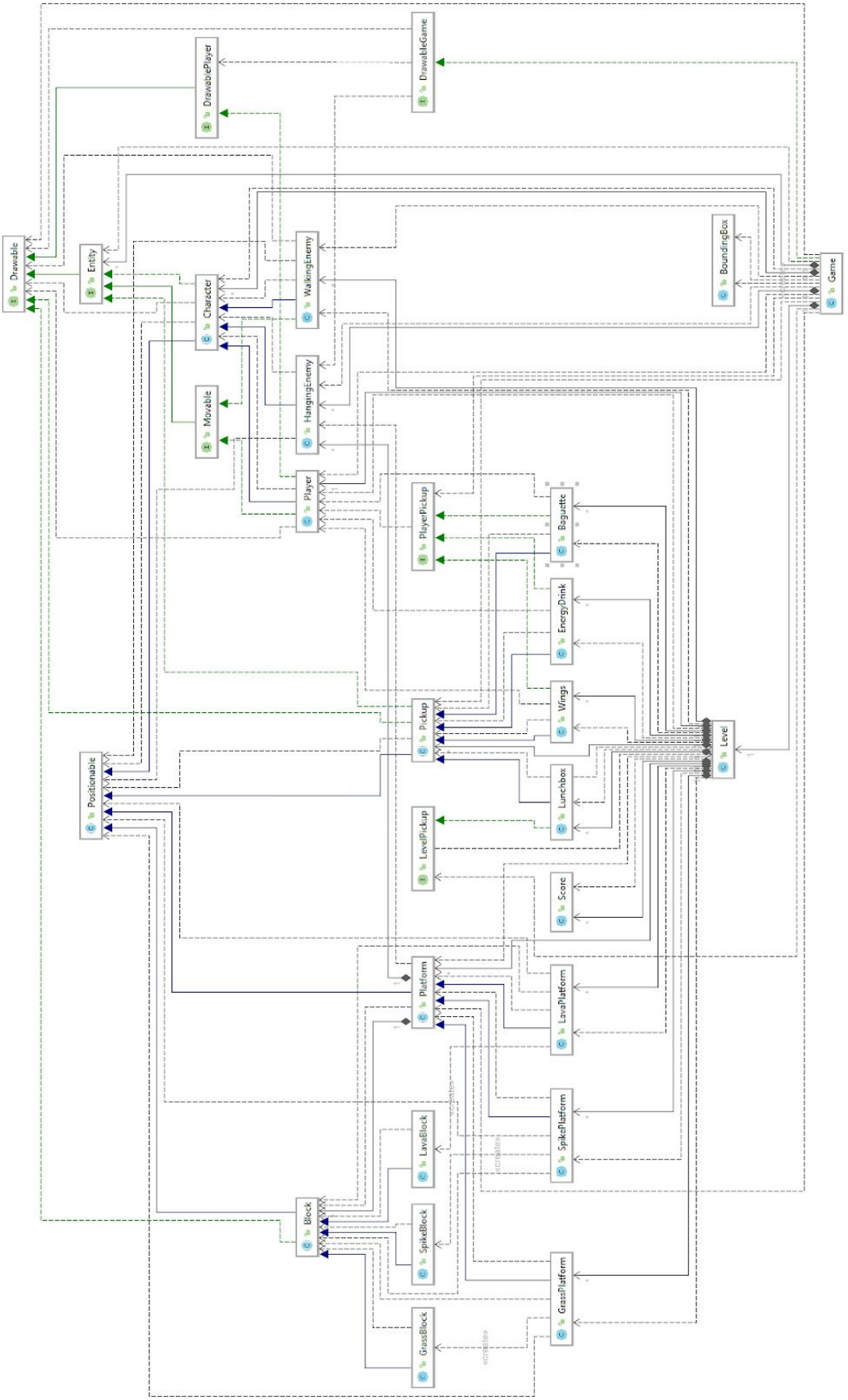
2.3 Game logic

The user can finish a level by eating all of the lunchboxes. A timer is keeping track of how long it took to finish, so that users can compete against each other and themselves. The player character has a health of 4 and can take damage from different objects in the game. When the health is 0 or if the player character falls below a certain Y-value, the player character dies and needs to restart the level in order to win again.

2.4 Package diagram



2.5 UML diagram of the model package



3 Subsystem decomposition

Packages

Model package

Contains all the classes describing the domain model. Does not have any LibGDX dependencies. Only has dependencies to util and service packages.

It contains interfaces with a Drawable-prefix, to enable exposure of sufficient information to the view. The abstract class Character holds code for colliding with blocks and getting damage. Both the player class and the enemy classes implements this. The enemy class doesn't take damage as for now, but in this way it is possible in the future. The Positionable class is realized by all objects with a position in the game, i.e. enemies, blocks, pickup, player character e.t.c.

The entry point is the Game class that loads a level, handles all collision and keeps track of the progress for the user.

Level package
Inner package of the model containing all level related entities. This package contains a platforms package, which holds the code for different types of platforms, consisting of different kinds of blocks. There are abstracts classes for blocks and platforms, and classes realizing different types of these.

It also contains a pickup package, with an abstract class called pickup, and different classes realizing this. There are two interfaces, to allow for pickups to affect the player character or the level.

Finally, the model package contains classes modeling the enemies, the player character and the level. The level class holds entities of all other classes in the level package.

View package

Responsible for all graphics for the application. Has indirect dependencies on the model through interfaces provided by the model.

Game package
Responsible for rendering the playable game, such as the sprites and their animations and the health, time and number of lunchboxes.

Menus package
Renders the menus, including their animations, and listens for user input.

Modal package
Renders the modals and listens for user input, and calls methods of the GameMenu interface.

Control package

The control package connects the model and view packages. Carries the responsibility for playing sounds. Controls the player character through user input and enemies through logic.

Menu package
Contain controller interfaces for all menus and modals, to enable navigation between them and the game.

Util package

The util package only contains a Timer class for now. The Timer is used for formatting milliseconds into readable strings. It is also used for calculating play time and getting high scores. The package is supposed to be independent from the context of the project.

Service package

Contains a parsing service for parsing levels and level info.
Note: This package has circular dependencies on the model and view package. This is because it parses JSON files to objects by using classes from these respective packages. To be able to have this service package rather than having this technical code inside, this was necessary.

4 Persistent data management

In the application, there are quite a few different data to be saved as files on the local hard drive. These include:

- The levels in JSON-format together with its meta-information
- Sprites and textures
- Sounds and music

All of these files are only read from the application. They are read using either LibGDX tools or the java.nio.Files library. The loading of levels is done when the users starts playing a level and all the sounds are loaded when starting the application, i.e. viewing the main menu. The levels and level meta infos are parsed from JSON files into Java objects from the Parser class. // Borde vi nämna att det är därför konstruktorerna ej används?

The only files that are written to are the level meta files. These contain only the name of the level and a list of Highscore objects. The Highscore object contains a time (score) and a name. When the user has set a new highscore, a new Highscore object is created with the

name and score of the user and appended to the list of Highscores in the level meta info. It is serialized into a JSON object then the user has entered her or his name.

7. Use cases

The following use cases are present in the application.

The user can:

- Navigate in the menus
- Start game // här innebär game inte riktigt samma som i definitionen längst upp i dokumentet...
- Select level
- Start level
- Complete level
- Quit level
- Quit application
- Make the player character move
- Make the player character jump
- Make the player character pick up an item

6 References

Stan documentation http://download.stan4j.com/app/stan-app-help-2.2.pdf
LibGDX documentation:
https://libgdx.badlogicgames.com/documentation/help/Documentation.html