System design document for Candy Monsters

Table of contents

Version: 1.0

Date: 2013-04-25

Author: Martin Hermansson, Sara Wallander, Sara Johansson, Josefin Kvillert

This version overrides all previous versions.

1. Introduction

1.1 Design goals

The design must be testable. The model must not be tied to the controller and view in order to enable the possibility of changing GUI.

1.2 Definitions, acronyms, abbreviations

- GUI, graphical user interface.
- Java, platform independent programming language.
- JRE, the Java Runtime Environment. What's needed to run a Java program.
- MVC, a design pattern in which one separates the application code (Model) from the GUI (View) and the Controller.
- Solid ground, everything that the character can't walk, jump or fall through.
- Air, everything that doesn't collide with other things.
- Non-empty space, all things that are not empty, for example solid ground and walls the chosen item, the item that the character is standing near
  placed on the..., on the image (not on top of the character, target, etc.)
- World, the area of the screen which can be referred as the "game window" (the area where the character is able to move without moving passed walls)
- Wall, included in solid ground

2. System design

2.1 Overview

The application will use the MVC model with some modifications. The view is updated by a method called in the controller. It is not using the Observer pattern. By using the library Slick2D things are painted out and by using the library JBox2D game physics are calculated.

2.1.1 The model functionality

Much of the functionality is built in Slick2D and JBox2D and isn't a part of the model. One example is the collision detection which is controlled by JBox2D. See also "2.1.8 Collision detection".
The model

2.1.2 Application basics

The game is a StateBasedGame in which many BasicGameStates are used. This is a way of changing views in the same window. There are three states: StartMenu, InGame and Highscore. By using an unique ID for each BasicGameState it's easy to switch view. The ID:s are stored as constants in the class Game.

2.1.3 Rules

The application will be a standalone, single-player, desktop based application. A single-player, platform game with a "collect and deliver" principle that will be shown in fullscreen. The game will have a set number of levels where each level has a time limit. Your mission will be to move around your character to collect items and deliver them to other characters in the world. When all characters have been delivered their specific item, you'll move to the next level. Your character has three lives during the whole game and these can be lost if your character gets hurt from the dangerous obstacles placed in the world. When all levels are cleared, and your character still is obtaining some life, you've finished the game.

2.1.4 Unique identifiers, global look-ups

See "The model functionality".

2.1.5 BlockMap

For making levels the program uses imported tmx-files containing references to pictures, properties of different tiles and so on. For each kind of tile the position is saved to an associated BlockMap. The solid ground tiles such as normal ground, ice, springs and ice springs are stored here. Moveable boxes' position, CandyMonster's position, Item's position, Spikes' position and Character's starting position are stored in those. The amount of time is also stored here as a property laid in a tile up in the left corner.

2.1.6 Controlling the game

The character is controller by using the arrow keys. To know if the character is supposed to move sideways the application checks if left/right key is held down. If so the character keeps moving. To pick up and drop down items KeyEvents are used to prevent the character from picking up and dropping down an Item multiple times.

2.1.7 Modified version of the MVC

A consistent feature is that the application creates all of its Model- and View parts in their respectively Controller. Because we use Slick2D some of the thoroughgoing controllers, representing different states, extends BasicGameState. Abstract methods in BasicGameState have to be overridden. The three methods are:

- init - executed once to initiate variables and so on.

- render - executed every time game is updating. Handles graphics update.
- update - executed every time game is updating. Handles updates that doesn't include graphics. Also have a parameter that tells the amount of time that has gone since last update.

Because of the render method and the way it works the application isn't using the Observer pattern. Instead the render method in each thoroughgoing controller forwards all parameters to a render method in the corresponding view which paints everything.
There is also another method that has been overridden in some controllers that extends BasicGameState. It is the method "enter". It is called every time a BasicGameState is entered. By doing this it's possible to control what's supposed to be created every time a new level is created and what shouldn't be created. It's also usable if the player is starting a new game when he/she is in the middle of a game. Some things have to be reset and it can be done here.
(See also 2.1.2 Application basics).

2.1.8 Collision detection

Because the application is an platform game collision detection is needed to be able to know if the character is pushing a moveable box, standing on it or on the solid ground or if it is traveling in the air. By using JBox2D and making bodies like boxes and other shapes and make them collide with each other it's possible to know whether the character or a moveable box is standing on the ground or is in the air.

2.1.9 Character's shape

A problem in JBox2D comes up when the squares (polygon shapes) are created by the positions taken from the TiledMap. Every tile becomes a JBox2D body and even though there should not be any height differences between them the character would stick in the gaps between the ground bodies by some reason. See figure 2.1.8.1 below.

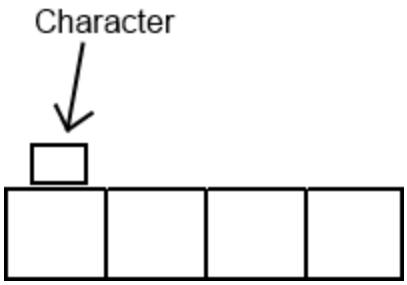

Figure 2.1.8.1. The way the character does not look like in the application's backend.

The problem is solved by making the character into a circle shape instead of a rectangle (polygon shape). Then it doesn't stick to the gaps. See figure 2.1.8.2 below.

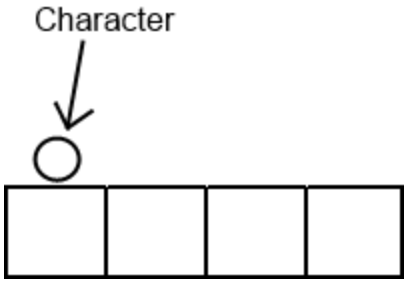

Figure 2.1.8.1. The character's body is formed as a circle shape in the application's backend.

2.1.10 Preventing character from walking on walls

As described in "2.1.8 Collision detection" JBox2D is used to know whether the character is colliding with other thing etc. JBox2D handles density, friction, restitution and so on but when the ground body is created the friction is relatively high because the character isn't supposed to slip around on normal ground.
The problem is that the whole ground tile have the same friction and so does the character if it's only one circle shape. If the character jumps towards a wall and a force is still added to the character's body the character will stick to the wall because of the friction. See figure 2.1.10.1 below.

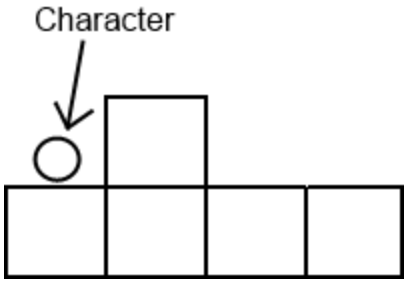

Figure 2.1.10.1. Because the character body and the ground bodies have high friction the character is able to stick to the walls if there is a force on the x-axis.

The solution in the application is made by two other circle bodies that are attached to the character's body. These have a lower friction and are of the same size as the character but are attached on the character body a little to the left-up and right-up measured from the character's centerpoint. See figure 2.1.10.2 below.

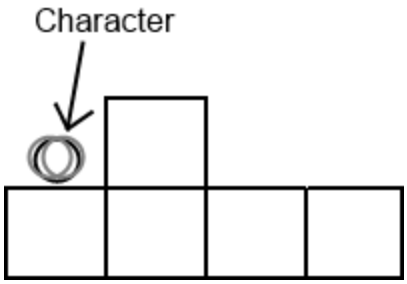

Figure 2.1.10.2. The character have two circle bodies attached on each side of itself that have a

lower friction to prevent wall climbing. This is how the character looks like in the application's backend.

By doing this the character isn't able to stick to the walls nor climbing them.

2.1.11 Score, life and time

The life and score remains over the whole game. When the game starts the player has three lives and the score is zero. As the player finishes courses the score will increase depending on the amount of time left and the number of items delivered. On the last level the remaining lives are playing a role on how much score the player will get. If there is enough score to get to the highscore board the player is asked to type his/her name and the score will be placed in the highscore (See also 2.1.12 Saving highscore).
The lives will be lost when the character is stepping on spikes. When the character does so it will become invincible for about 1 second and will also blink, so really the player will only lose a life when the character is stepping . The time 1 second is calculated by the parameter in the update method (see also 2.1.7 Modified version of the MVC). The time since last update is added to a variable until it reaches 1000 milliseconds. See 2.1.13 Spikes collision for more information on how to know whether the character is standing on spikes or not.

2.1.12 Saving highscore

There are two txt-files that are used to keep track of the highscore list. One is for the integer numbers that make the score and one is for the names the player types in after getting a new highscore. when the game starts these files are either created or read from and the values, every single score or name put on a new line in the document, are saved in an array with the size 10.
When the player reaches a game over or victory the final score is compared to the last score in the array. If the player have a high enough score the next screen will be the "New Highscore" screen where he gets to type in his name. Then the score is saved using the method saveScore(int newScore, String name). This method sorts and saves the score and name into the arrays and writes them to the files.

2.1.13 Spikes collision

To be able to know if the character is colliding with spikes, the application uses sensors included in and created by JBox2D. Every spike has a sensor to be able to know when the character starts to have contact with a spike and when it stops to have it. The controller class of the spikes implements the interface ContactListener where inter alia the methods beginContact and endContact are overridden. By doing this it's possible to know whether the character body is touching a spike or not.

2.1.14 Menus - navigation

The menus are simply illustrated by rectangles and labels. Invisible rectangles are drawn on the screen and labels consisting of images are drawn on top of these. To illustrate movement by

pushing the arrow keys, either up or down, the rectangle marked is filled. Consequently, it looks like buttons being marked in the order you presses the arrow keys. A for loop creates both labels and rectangles and a simple int keeps track of which rectangle should be filled. Since the menus are very similar in both their appearance and functionality, abstract classes was created for them to extend with code they had in common. When pressing enter, a switch keeps track of each button's own functionality.

2.2 Software decomposition

2.2.1 General

- model: the model part of MVC
- controller: control classes for the MVC model
- view: view parts of MVC
- utils: conversion methods, I/O

2.2.2 Decomposition into subsystems
There are no subsystems. Only a utils package including classes which is of assistance to other classes.

2.2.3 Layering

See Figure 2.2.4.1 below.

2.2.4 Dependancy analysis

Dependencies are as shown in the Figure 2.2.4.1 and Figure 2.2.4.2:

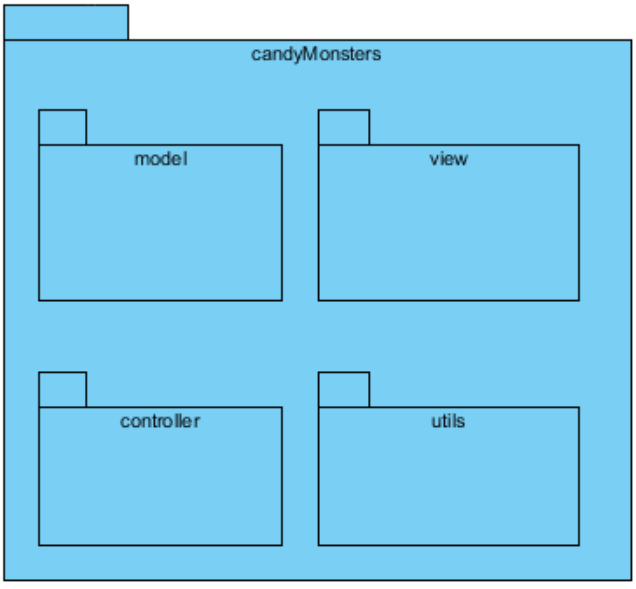

Figure 2.2.4.1: High level design

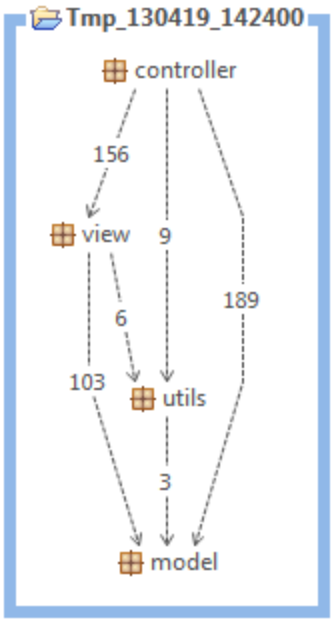

Figure 2.2.4.2: STAN Diagram, Layering & Dependencies analysis

2.3 Concurrency issues

By using Slick2D thread controlling update method and render method are started. There are no concurrency issues though.

2.4 Persistent data management

Files that are loaded into the application is pictures and TMX-files which contains the TiledMaps. See also “2.1.4 BlockMap”. Pause picture is saved to harddrive to be used when the game is paused.

2.5 Access control and security

NA

2.6 Boundary conditions

Application is started and exited as a normal application.

3. References

MVC, see http://en.wikipedia.org/wiki/Model-view-controller