System design document for InfinityRun

Version: 1.1

This version overrides all previous versions.

2017-05-28

Authors:

Leo Oinonen

Mikael Törnqvist

Jacob Rohdin

Anders Bäckelie

1. Introduction

This is the system design document for the Java game InfinityRun.

1.1 Design goals

This document describes the construction of the InfinityRun application specified in the RAD document.

Design goals:
Be able to switch the graphical API (libgdx), without rewriting the entire application.
Extensibility, simple to create new enemies or add new “blocks” into the world. It should be easy to write a new class that extends the functionality of its parent.
Make use of the command pattern in upgrades to be able to add or edit upgrades easy later.
Be able to test the critical functions with JUnit.

1.2 Definitions, acronyms and abbreviations

Technical definitions

- **GUI** - graphical user interface, like menus.
- **Java** - platform independent programming language
- **JRE** - the Java Runtime Environment. Additional software needed to run a Java application.
- **AI -** Artificial Intelligence; in other words entities that are programmed to exhibit situation-dependent behaviour.
- **MVC** - a program modeling pattern to separate the view, controller and model. The point is that you can replace one of these three parts without affecting the other two parts.

Game Definitions

- **HP** or Health Points, the total number of damage something can take before dying.
- **Movement Speed** is the speed which things can move.
- **Coins** which is the currency of the games, used for purchasing upgrades and similar things. The number of coins looted can be increased with upgrades also.
- **Looting** determines how many coins you will get per coin object you pick up.
- **Rooms** are the created rooms that the game generates the world with, the world has many rooms which are randomly determined by an algorithm.
- **Death** when your HP reaches zero, you will then be transported to the shop. This only means you have to start from the starting point again - your character will not be removed.
- **Shop** is the place where you can spend your coins for upgrades or weapons.
- **Upgrades** will be making your character better through different weapons, more HP, more jumps, higher jumps, higher damage, more critical hit chance and better looting.
- **Score** will be measured in number of rooms cleared.

- **Damage** is the number of HP something can deal in one hit.
- **Critical Hit Chance** is the chance you have to do a special attack that will deal increased damage depending on your *critical hit damage*.
- **Critical Hit Damage** increases the damage of your critical hits. Without any critical hit damage you deal 2 times your normal damage.
- **Monsters** will have some sort of AI, they can be killed if you have enough damage or outrun with enough movement speed and flexibility.
- **Character** is the one you as a player control, you can make the character stronger by collecting coins which later are used for purchasing upgrades.
- **World** is an infinite puzzle of rooms, each room's entrance is put together with another room's entrance which makes for a never ending world with unlimited exploration.
- **Air Jump** is a jump performed in the air, you will be able to do double jumps and even more jumps depending on your upgrades.

2. System Architecture

The Infinityrun game is a single player, desktop application for one user. It requires no network connection. Everything is launched from the application.

The Game application that the project depends on requires an advanced kind of OpenGL wrapper / DirectX wrapper. Because of the requirements of the game, a frame based framework is needed, as opposed to an event based framework such as Java swing. The application is based on a model-view-controller pattern, but the view and the controller is only slightly separated due to the constraints of a frame based framework.

Most of the model will be based in the World class, into which all WorldObjects will be loaded once it has received knowledge of whether it should use a new character or if an existing character will be loaded in. It is however important to note that the World class itself does not load any data from storage, it merely receives it from dedicated I/O service classes. GameScreen takes information about all the WorldObjects in World and draws a graphical representation of them. A controller also listens for specific keyboard input in order to tell GameScreen to load the ShopScreen or the PauseMenuScreen. Depending on the framework the controller may be a part of GameScreen as a combined View/Controller. The PauseMenuScreen has the ability to unpause, i.e return to the GameScreen, the ability to return to the MainMenuScreen, as well as the ability to call upon the saving service class (writing character data to a textfile) and then exit the game.

The flow of the program is such that at the beginning of each frame it will collect all the input with help of an input class, then the World instance will frame every WorldObject that the world contains with the Input state along with delta time for this frame. Every WorldObject has it’s own responsibility, they can check for Collision if they need or use the input state if they need. LivingObject extends WorldObject and this class represents living objects in the world like the Character and Enemies. LivingObject contains a list of upgrades that it has and the upgrades is built like a command pattern, each frame LivingObject will call frame on the upgrades to get the benefits that they do. Since this is a game the model is huge while the view and controller are quite contained and small. The classes that are part of the view are the various screens in the Screens package. The classes that are part of the controller are the IInput interface along with the different classes that implement it, while GUI is part of both the view and controller. The model is the rest of the project classes.

2.1 General observations

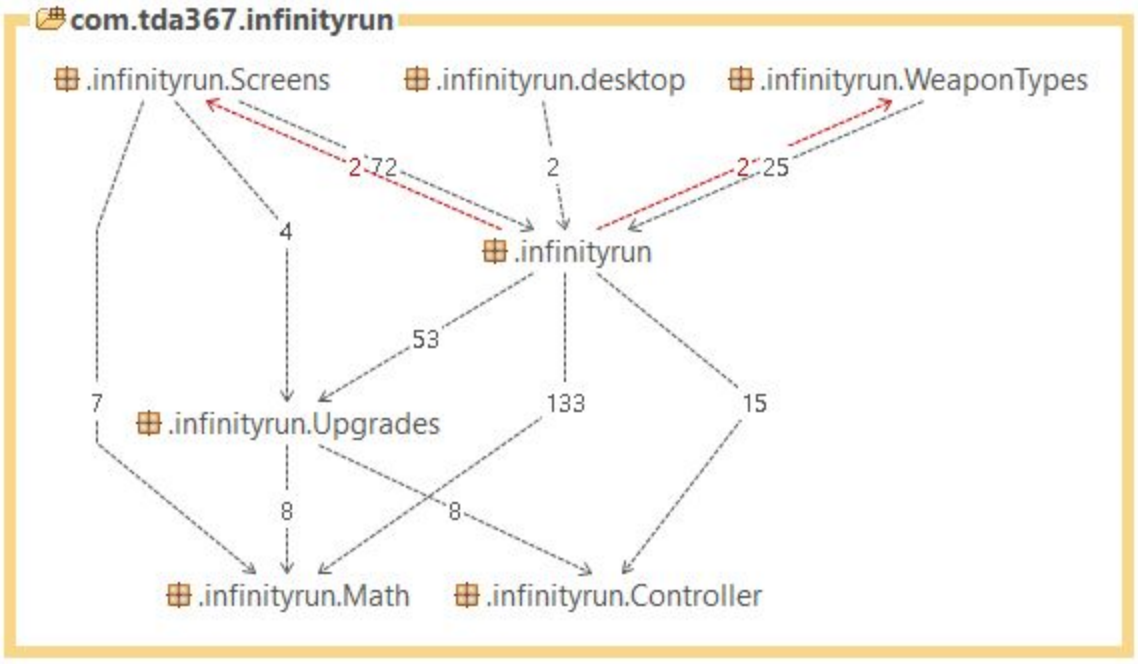
Composition
LivingObject has multiple Upgrades.
World has one IInput.
LivingObject has one Melee Weapon.

Libgdx
The API is exposed to several screens for rendering, it is also exposed to InputGDX that derives IInuput to collect input.
Then all of our GUI classes expose Libgdx since they are directly tied to this API.

MVC
It is quite complicated to make a good MVC pattern in a frame based application. The default way to implement MVC is to route events to different parts of the code with help from interfaces. Since the framework is frame based we can't route input somewhere so we just collect them in a specific class and then use a struct/class to pass the incoming states to the classes and let them handle what to do.

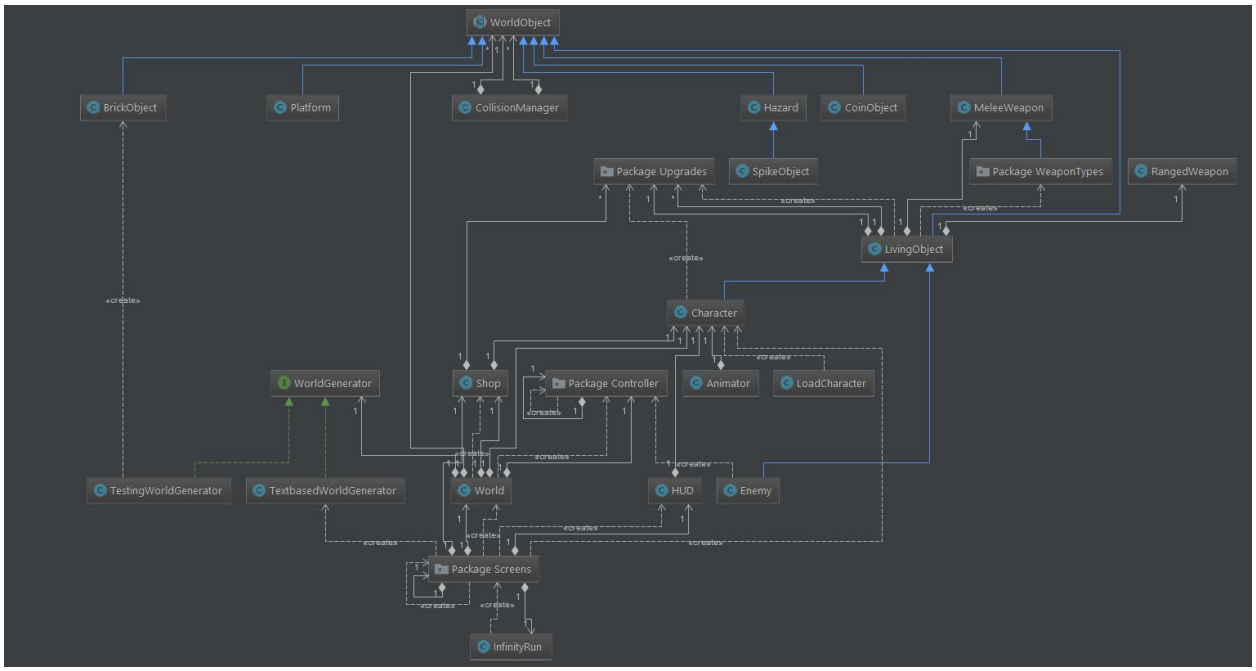
Extensibility
Since extensibility in the model is important, the project has been written to easily extend enemies and/or make new ones, as is desired. This would allow for enemies with different AI behaviour. The model for enemies can be extended to new enemies by overriding the "Frame" method that is the "Tick" for the enemy. The corresponding extends for new tiles or upgrades are also very simple, meaning that it's very easy to expand the project with new features.

Package Dependencies
We have approximately 1% tangled code but we decided that we will keep it so since it is a lot clearer to have WeaponTypes in a different package and the same for the Screens package. The only thing that we need to change in our project to get to 0% tangled code is to remove those packages, but as mentioned we won't.



UML

The UML diagram is generated by IntelliJ and the result is quite good. We have removed the Math package from the diagram to make it clearer since every class use something from the Math package.



3. Subsystems decomposition

Kd-tree is the subsystem for finding points in 2d-space. This allows for fast range search between WorldObjects, collision and identifying the internal position of objects.

3.1 Core

Core contains most of the project, primary the model and also some parts that contains libgdx since we build this on top of libgdx. The parts that are not in core are parts that doesn't have any dependencies to it. The WorldObject hirarcy is placed in core and everything that extends WorldObject, like character and enemy. WorldObject represents everything that can be seen on the screen and contains the functionality of how they work. How the WorldObjects are drawn is however not placed in them themselves since this is not part of the model. Core also contains the worldgeneration.
Worldgeneration is done by an algorithm that uses 15 different room patterns. Each of these patterns represents an unique set of the four exits the WorldGenaration is based on. (up, right, down, left.) The algorithm then makes sure to only connect an exit to an exit and a wall to a wall to. This can be seen as a reversed puzzle by connecting matching pieces.

The WorldGenerator then translates a room with the matching pattern from a textfile containing an ASCII representation of the WorldObjects and returns this to the world.

3.1.1 IndexPoint

Indexpoint overrides the coordinates with a unique hashcode for storage in the TextBasedWorldGenerator class. This is to guarantee that each room has an unique identifier in the list so that two rooms are not stored in the same space.

3.2 Math

The Math package contains helper classes with math.
This includes: Vec2, Vec4, Rect and Utils.
Vec2 is a 2 dimensional coordinate class with some methods to mutate the data.
Vec4 is used for storage of a 4 dimensional point in fringe cases.
Rect will represent a rectangle, this project represents everything as rectangles so this is a very good way to get intersection and rendering positions or size.
The Utils class has helper methods to determine distance both internally and externally for WorldObjects.

3.2.2 KD-Tree

We have implemented our own collision system, if we did this the naive way like brute force collision we would have lags and the game would not be playable. To solve this

problem we have created our own KD-Tree to find the K nearest neighbors with O(log(n)) complexity. We have also created a method in the KD-Tree called rangeSearch2d, this method is used to get every object inside a rectangle so that if we wanted to find everything that we could hit within a rectangle, this method would return everything inside. The KD-Tree is built on the node pattern instead of the List pattern so we are required to have a helper class named KDTreeNode. In the KD-Tree we will store the point that represents the node and also a generic type of what this point represents so that we can easily get back the 10 nearest worldObject if this is what we are looking for.

3.3 Screens

The package Screens contains different screens that can be viewed in different situations like the MainMenuScreen and the PauseMenuScreen. They are responsible for drawing various parts of the game, as well as being responsible for listening to key or button input in certain cases. This is because of the sometimes unfortunate way certain frameworks work.

3.4 WorldRooms.txt

WorldRooms is a textfile that represents rooms as letters. This is so that it is easily modifiable by humans. A room is 25 letters wide and contains 15 rows, and is seperated by the Letter combination that designates what room pattern the room represents.

3.5 Upgrade

Upgrade are a class for all different upgrades that the LivingObject can have. We have for example JumpH and Speed which makes it possible for one specific living object to move and jump. Upgrades also holds things like critical hit chance and critical hit damage which later are used in meleeWeapon

3.5 WeaponTypes

WeaponTypes contains classes that represent different weapons. One movableObject can have one melee weapon and one rangedWeapon to use when it is attacking. The parameters you send in a meleehandling which is a multiplier to your base damage, critical hit chance which will increase the chance of making a critical hit and critical hit damage which will increase the damage a critical hit does.

3.6 Controller

The package controller contains classes that are required to get input each frame so that each worldObject can interpret the inputs themselves. Since these classes are not part of the model and doesn't have any dependencies on the core package, we decided that it is best placed in a separate Package.

3.7 JUnit Tests

The Project also contains a testing enviroment for singular tests of core features to ensure that they work properly. Added features should also be tested here to guarantee they work as intended. These tests are implemented with Junit.
In descending order, the current tests are:

testDamage - asserts that livingObjects can take damage.

testCoinCollection -asserts that the Character can collect coins

testPurchaseUpgrade - asserts that the Character can purchase upgrades if it has enough coins

testRegeneration - asserts that a LivingObject regenerates damage over time.

testSpeedUpgrade - asserts that the movespeed is increased when upgraded.

testJumpLvl - asserts that the jump height is increased when upgraded

testUpgradesExist -asserts that all upgrades are initialized to default when creating a new Character.

4. Persistent data management

The persistent data is generally handled in one of two ways. Textures are stored in folders describing their area of use e.g. “WorldObjects” or “HUD”. In some cases the textures are put together into one image form a so called “Texture Atlas” together with a pack file that describes the positions of the individual textures within the image. The textures are all stored in the format .png. The rooms in the game are saved in WorldRooms.txt. The character saves are saved in savedata.txt. This file is created when first saving a character.

5 Access control and security

None. This program runs locally and is Launched/exited like a normal application.

6 References

1.
Libgdx
https://libgdx.badlogicgames.com/
2.
MVC, see http://en.wikipedia.org/wiki/Model-view-controller