System design document for Sniper-Monkey

Elias Falk Dadi Andrason Vincent Hellner
Kevin Jeryd

2021-10-24

1 Introduction

This document is a description of our software application's systemic design properties through displaying and explaining system architecture, system design, data management and quality control. The application, Snipermonkey, is a 2D platform fighting game with a slight variation to the standard concept. The alteration is a game-feature where each player needs to choose two fighters to compete with. By combining two fighters with different quirks and abilities, the players are able to make adaptions to their fighting-style in the middle of the game by switching fighter and countering their opponent which gives more depth to the game.

1.1 Definitions, acronyms, and abbreviations

- MVC
  - ”Model, view, controller”. Architectural design pattern for dividing your application.
- Texture
  - An image.
- Sprite
  - A section of an image.
- Sprite sheet
  - A texture including multiple sprites arranged in rows and columns.
- Tile set
  - A sprite sheet but for tiles.

2 System architecture

The system is built around the architectural design pattern MVC. The pattern divides the program logic into three related elements.

- **The model**: contains classes with dynamic data and logic such as Player, Collision Engine, Fluctuating Attribute etc.
- **The view**: contains the classes that make up our graphical user interfaces.
- **The controller**: contains classes that takes data as input and converts it to commands for the model or the view.

The view heavily relies on the libgdx library to render the application. The controller also use the library to figure out the time between frames in order to update the game correctly.

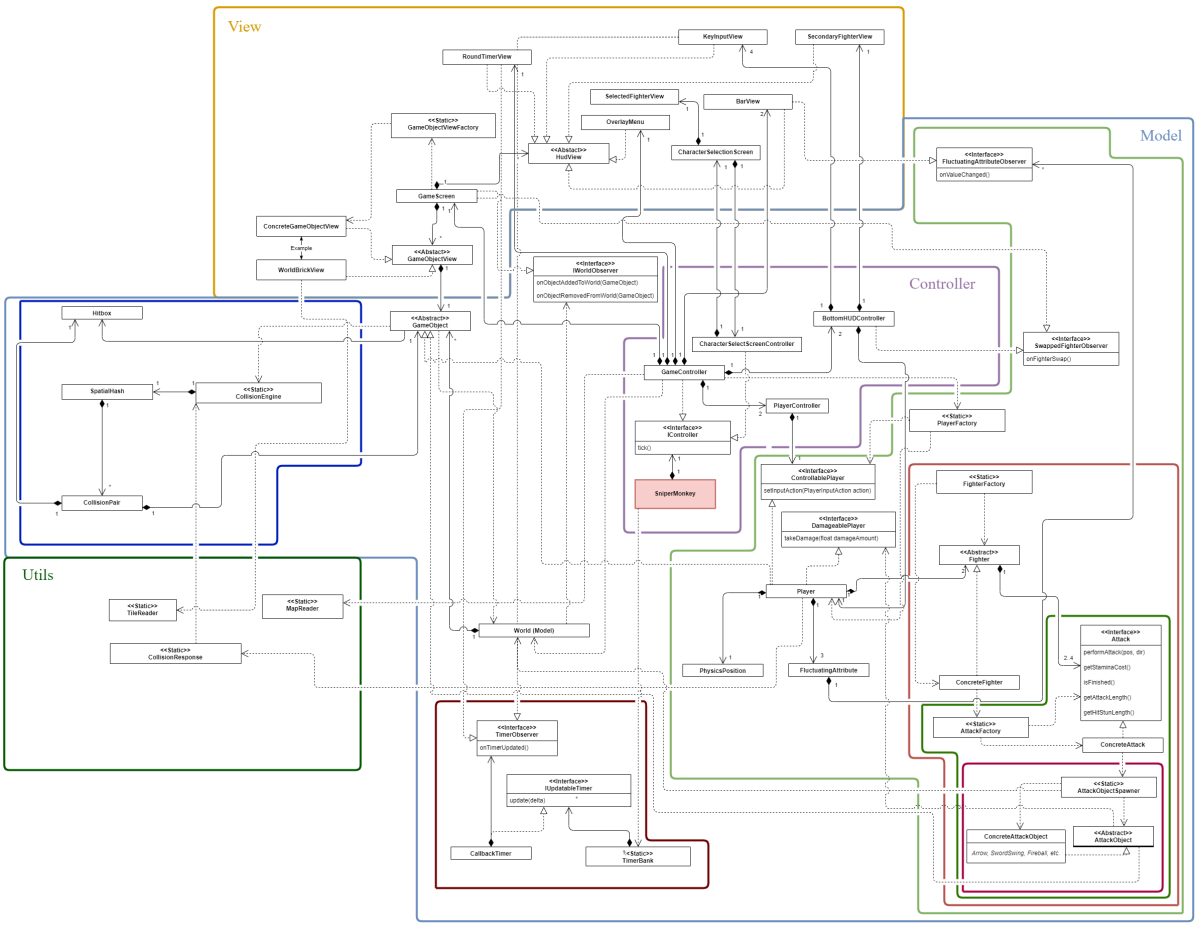
2.1 Flow

The general flow of the application is as follows.

1. The SniperMonkey application is initialized.
2. The SniperMonkey application keeps track of the current ScreenController.
3. The ScreenController is set to a new CharacterSelectionScreenController.
4. The current ScreenController is updated every frame.
5. The current ScreenController (CharacterSelectionScreenController) keeps track of and lets players choose all their fighters.
6. Once the fighters have been choosen the current SniperMonkey ScreenController is set to a new GameController and the choosen fighters are supplied.
7. From here the map and players are initialized.
8. The current SniperMonkey ScreenController (GameController) continues to be updated every frame and the game is played until the time runs out or one player has been killed.
9. If the players wish to restart the game the flow once again moves to step 3.

3 System design

3.1 Design Model



3.2 Model View Controller

This application implements the traditional MVC pattern by dividing the application into the three components Model, View and Controller. The application also makes heavy use of View models for GameObjects which all have seperate views.

The main controller of the application is the class SniperMonkey which initializes the GameController and handles the updating of the application itself. The GameController is then in charge of initializing the game model and the game screen view and ties all of this together by mostly registering various observers.

The observers between the model and view is used extensively as a means to communicate where on each notification of a change a View model either re-reads data using its reference to a model-object or the data is sent to the view in the notification itself.

3.3 Design Patterns

3.3.1 Singleton

The singleton pattern's purpose is pass a single instance of a class around to many different classes that need to use that same instance. The singleton pattern is used for the game world which contains all the game objects used in the round such as players, projectiles and platforms. The game world then has the responsibility to update each game object each frame. The purpose of using this pattern is to easily add and remove game objects while running the game while only depending on the world class.

3.3.2 State Pattern

A version of the state pattern is used in the player class which determines what the player can do in a given time. The player has two states, a movement state and an ability state. These states are implemented as method references, where the current state is called upon on each update. Each state determine what and when the next state will be and what inputs are allowed to influence the player, for example if the player can jump or not.

3.3.3 Factory Pattern

The factory pattern is used to create both players, fighter and attacks but is more thoroughly used by fighters and attacks. This pattern is implemented to hide the creation logic of creating a player and fighter and returning the correct abstraction.

3.3.4 Observer Pattern

The observer pattern is used throughout the application. It involves using a subscriber based system where observers can be registered to a subject which notifies them when necessary. In the program this is used to avoid unnecessary dependencies and the removes the necessity of having objects ask other objects for data constantly. Most notably this is used during the creation of GameObjectViews which are created by the GameRenderer, an observer of the World. Once a GameObject is added the the World the GameRenderer is notified and proceeds to create the view.

4 Persistent data management

The application stores many different resources such as images, maps, fonts, skins and configuration files. All resources are found in an assert folder with corresponding sub folders for each different kind of resource.

Within the images folder there exists different texture packs for fighters as well as map tiles and backgrounds. The fighters are formatted such as each animation has their own sprite sheet and the different map tiles are all found in a single image for each map texture pack.

The map data is structured in a way that each map have their own folder within the map folder representing the name of the map. Within the specific map folder there are then two files one for the tile set data (.tsx) and one for the map data (.tmx) which places the different tiles from the tile set drawn in a program such as Tiled.

Configuration files are found in the cfg folder where there are different configuration files for different types of data such as the values of a player or game data. Each file consists of key value pairs in a KEY=VALUE format. An example would be MAP_NAME=grass map 2.

Different fonts simply have their own folder within the fonts folder where all the font files for that font exist. For example the font Roboto-Black.ttf is found at ”fonts/Roboto/Robot-Black.ttf” starting from the assets folder.

Skins for buttons and other elements used by libgdx are stored in their respective folder within the skins folder. Within each specific skins sub folder all the relevant files exists for that skin such as the images and fonts and atlas file.

5 Quality

5.1 Testing

This application mainly tests the part of the program that represent the model within the MVC pattern. All tests are performed using JUnit[1] and can be found at "core/src/test" within the source code.

5.2 Known issues

- Resizing the window while at screen Y and going to screen X from screen Y scales the UI elements differently from simply scaling screen X.
- Having an unreasonable low framerate (<1 frame per second) will make players phase through the tiles.
- Using an attack while moving, falling or jumping will result in the attack object to spawn at the place where the player pressed the attack, not where the player currently is. For example if an arrow is shot while in the air the arrow won't shoot until the player has fallen to the ground and the actual arrow will spawn in the air, where it's not suppose to. This is due to a spawn delay of the attack object which doesn't take into account the player's movement speed.
- Spawn points y values are flipped which results in having to place the spawn positions mirrored in the Tiled program.
- Currently when blocking, the fighter's animation is set to dying and should instead be something else that represents a blocking fighter.

5.3 Analytical tools

5.3.1 STAN

Due to the fact that the executable found on stan4j.com[2] not being able to run, there is no analysis regarding the code's structure.

5.3.2 PMD

To analyze the quality of the code the tool PMD[3] was used. The rulesets that were applied while running the program was design, error-prone, multi-threading, performance and security. The result and ruleset can be found on github at "analysis/PMD/".

5.4 Access control and security

The application does not use any kind of access control and simply runs locally without any need for user authentication or authorization.

References

[1] JUnit, Version 4.13.2, [Software], Unknown : JUnit, 2021.

[2] STAN, Version 2.2.0, [Software], Weilrod, Germany : Bugan IT Consulting UG, 2021.

[3] PMD, Version 6.39.0, [Software], Munich, Germany : PMD, 2021.