



UNIVERSITY OF LIÈGE

INFO2051

Object-Oriented Programming on Mobile
Devices

Santa Clash : Report

Vasco AGNELLO
Adrien CHIDO
Mehdi TESTOURI

December 16, 2019

1 Introduction

In this report we will present the application we developed as part of the INFO2051 course. We decided to create a smash-like game, we will describe how we build the application in the following sections as well as its structure.

1.1 App Description

Santa Clash is a fighting game inspired from the *Super Smash Bros* games of *Nintendo*. The game is a platformer where two players are fighting each other, the goal of a player is to eject his opponent behind the edges of the map they are playing on. To do so, the players have a set of abilities accessible through the use of several buttons.

1.1.1 Buttons


- The button **A** is used to throw a punch in the orientation of your character and deal damages (see 1.1.2) continually in a small area.
- The button **B** is used to do a circular kick in the orientation of your character and deal a few damages continually. Moreover, any character that is being hit by a circular kick will be propelled in an opposite direction from the kick following an oblique trajectory.
- The **fireball** button, is used to throw a projectile horizontally at a constant speed that deals a great amount of damage.

In addition, the player is able to move in his environment with the help of three directional buttons that allow him to go left, right or to (double) jump.

1.1.2 Damages

The overall damages of a character is given by a percentage 0.00% that can grow up to 300%, the higher the damages are, the stronger the propulsion of a circular kick is. There is no way to decrease your damages but by dying (see 1.1.3).

1.1.3 Lives

The lives of a player, just like the damages, are displayed at the top of the game screen  and each player is granted three of them. The objective is to take your opponent down to zero life, if you manage to do so, you win the game immediately, if he does it first, you loose. Being expelled out of the map's borders kills you, dying reset your own damages.

1.2 App structure

As shown in the Figure 1 the app is structured between menu pages that guide the user in its navigation.

First, there is the homepage which is the starting point of the application. It shows the title and a little animation of two Santas fighting. A blinking "touch screen" is also present.

By clicking anywhere on the screen the user falls in the first menu page. On this menu the player can click on 4 buttons. The training button which launch a non-multiplayer game where the opponent is immobile. The purpose of the training mode is to let the user familiarize itself with the commands and mechanics of the game. The multiplayer button which lead to another menu page that will be discuss later. The credits button which lead to a scrolling text that thank the developers and the assets drawer of the game. And finally a return button to come back to the home screen.

If the player select multiplayer, he arrives on a second menu page where he can select create a game or join a game.

The create game button leads to a selection of character (red or green Santa) and after, a selection of the arena (snowland or sunnyland). After the selection the user is directed to a page where the app wait for an other mobile device to join the game. On this screen the user has the possibility to return to the second menu page thanks to a return button.

If the player chooses to join a game, a prejoin page remind him that he needs to first pairs his device with the opponent's device before joining a game. Then the join page let the user select his opponent, refresh the page or return to the second menu thanks to several buttons. An easter egg page is also dissimulated in this page, indeed by clicking the right Bluetooth icon ,

the user arrives on a page where several "Did you know ..." type information are present.

When two mobiles devices connect themselves together a multiplayer game start. At the end of the game, the players arrive on the end screen page which informs them if they won or not and then go back to the first menu page.

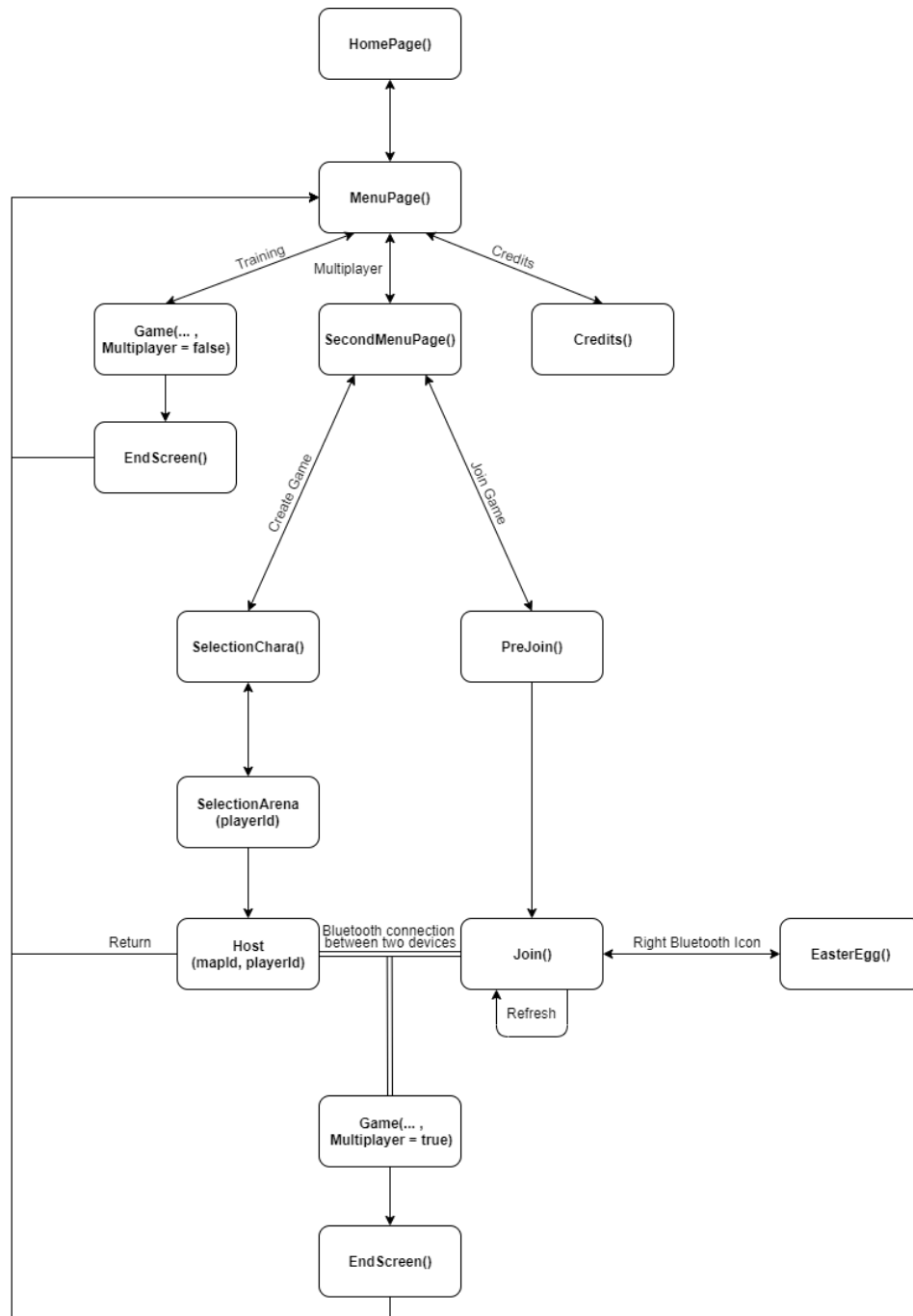


Figure 1: Application structure.

2 Technical

2.1 Code Structure

The code is structured around three main categories :

- The menus.
- The game engine.
- The game logic and assets.

The **menus** mainly contain the widget classes use to define the different menus of the game such as the character selection, the arena selection, the game modes (training and multiplayer) and finally, to launch a game session.

The **game engine** implements the different modules needed to make a game running. The main goals are :

- Provide a framework to make it easy to develop a game which means providing the required class to able to interface with the engine (for instance, the **Asset** class).
- Provide a game loop with an implementable game logic that is responsible to update the different game assets prior to the physical update and visual rendering.
- Provide an interface to detect and register the player inputs.
- A game engine should be agnostic of the actual game it is running and hence it must be reusable.

The **game logic and assets** contain the implementation of the aforementioned game logic with its required assets. Unlike the game engine, the classes implemented here can be **game specific**.

2.2 Menus

The menus are separated into multiples files in order to have a clear structure of the roading between them. A menu page of this app is usually a Stateless Widget whose build function return a scaffold widget. Note that the return button of the phone is disable on all the menu pages. Indeed return buttons have been made in order to go back to the preceding menu page.

2.2.1 main

The *main.dart* file is the entry point of the app thus the homepage is localised in this file. It consist of a title , an animation made with a stateful widget that switch between different assets in order to form the movement and a blinking text which is also a stateful widget that uses the opacity of the text in other to give the blinking effect.

2.2.2 menu

The *menu.dart* file is composed of the two main menu page of the game, the first menu which let the user chooses between training, multiplayer and credits. The second menu page handles the choice of joining or hosting a game.

The buttons are made thanks to clickable images which have been specially design for this app.

2.2.3 credits

The *credits.dart* file is composed of the credits of the app. The credits are made thanks to a scrolling text implemented with the *SingleChildScrollView* widget.

2.2.4 host

The *host.dart* file is composed of the character selection, arena selection and host page. The character selection is composed of two clickable container where each Santa is displayed with a little animation that is made with the same logic as the homepage animation. After the selection of the character the arena selection page ,which take in argument the previous choice of character, is displayed. Then the host page is made in order to wait for another

device to join the game. That is possible thanks to a stateful widget which loop until a connection. That widget takes in argument the map id and the character id that has been chosen by the user.

2.2.5 join

The *join.dart* file is composed of three pages. The first page remind the user that he needs to pair his device with the opponent before joining his game. The second page is the actual join page where you can select your opponent thanks to a clickable scrolling text. Two pop up windows have been made in order to confirm the selection of a device in the scrolling list and to let the user know that the connection failed if the Bluetooth connection did not work.

An easter egg page is also hidden in the join page. This easter egg page is only composed of a scrolling text.

2.2.6 endscreen

The *endscreen.dart* is only composed of a page that change with its argument. If the argument is equal to the const VICTORY , a victory screen is return and the same logic is applied if the argument is equal to the const DEFEAT or the const CONNECTION LOST.

2.3 Game engine

This section provides a general description of the game engine and its components. A global overview of the engine structure is illustrated in figure 2.

2.3.1 SmashEngine

This is the top class of the engine and also a widget so that it can be easily inserted in the widget tree of the app. It extends an inherited widget to make it possible for the input gestures to send the registered inputs to the renderer.

Upon creation, the engine must be provided with game designer implemented `gameAssets`, `inputGestures` and `gameLogic` classes specific to the game.

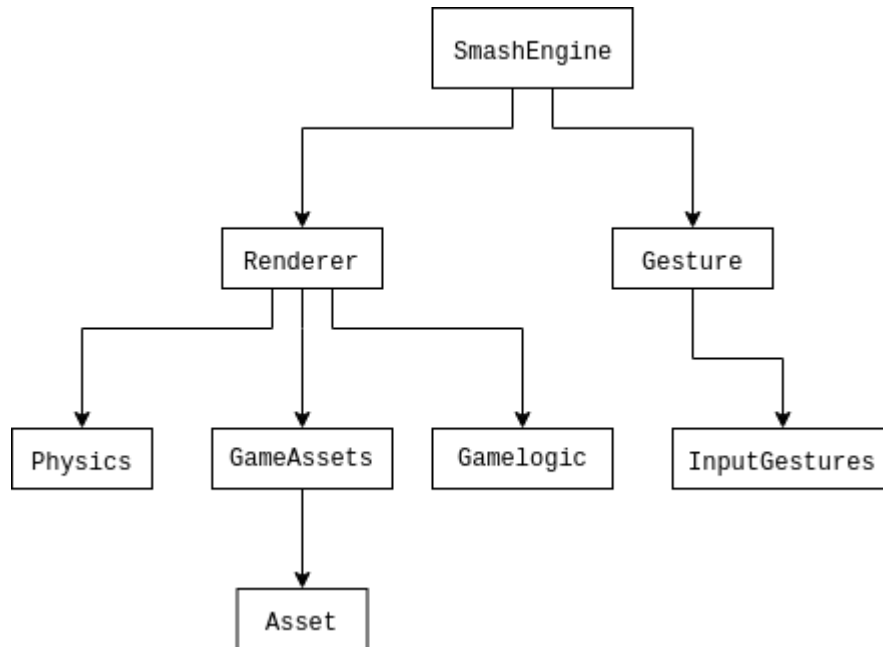


Figure 2: Overview of the engine structure.

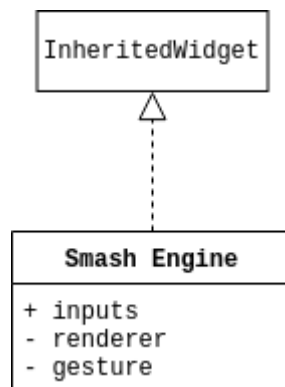


Figure 3: Smash Engine top class.

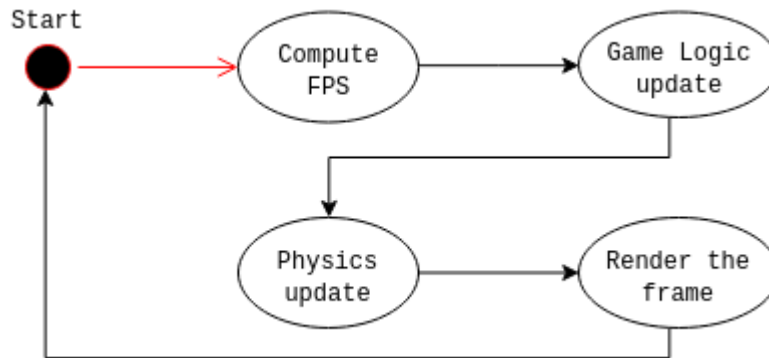


Figure 4: Game loop.

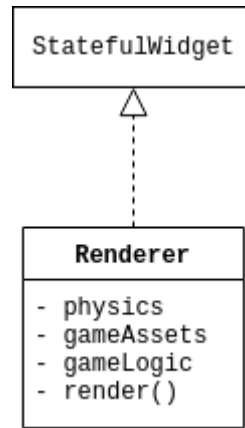


Figure 5: Renderer.

2.3.2 Renderer

The renderer is the class responsible for the rendering of all assets of the game by implementing the game loop (illustrated on figure 4). The renderer is a stateful widget that basically contains an animation that loops indefinitely for which a listener is attached and this listener (**render** method on figure 5) is what provides the game loop.

The renderer also compute the FPS (*frames per second*) which are used by the physics to correctly update the physical assets.

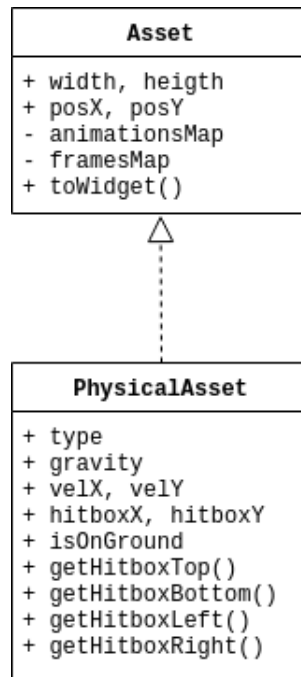


Figure 6: Asset.

2.3.3 GameLogic

The **GameLogic** is an abstract class whose children must implements an **update** method that is called by the renderer in the game loop.

The game logic is used to model the rules of the game and to update the assets accordingly. It can, for instance, read the inputs and decide their effects on the assets or check if one character hit another, etc.

2.3.4 Asset

The **Asset** class is the core object of the engine. It is used to represent any element of the game from the characters to the background image. An asset object can be converted to a **Widget** object thanks to its **toWidget** method which is used by the renderer to display the assets.

There are two types of assets : the **assets** and the **physical assets** as illustrated in figure 6).

Physics
- currFps - gravity + update()

Figure 7: Physics.

Asset The *normal* asset is a minimalist and abstract asset class that only contains visual properties (image file, animations and position) but doesn't have a *physical* existence in the game. Those assets can be updated by the game logic but won't be update by the physics in the game loop (figure 4). These assets can be for instance, UI elements, background images, etc.

Physical asset The physical assets are asset that have a *physical* existence in the game which means that they will be updated by the **Physics** class in the game loop (figure 4) by, for instance, applying the gravity or detecting collisions. These assets can be characters, platforms, projectiles, etc.

Another important component of the asset is the **animation**. An animation is basically a succession of frames with a specific timing. The animations are also tied to an **animationCallback** method that can be overridden by the asset designer to implement more complex behaviours related to these animations.

By default, the assets don't have animations but the asset designer can easily add ones by overriding an **animationsFactory** method.

2.3.5 Physics

The **Physics** class is used to update the physical assets using its **update** method (figure 7) called by the renderer. This method basically, applies the gravity, detects and deals with collisions and updates the positions of the assets using their velocities and the FPS (provided by the renderer).

2.3.6 Gesture

The **Gesture** object contains (figure 8) all input gestures objects that are used to provide the player with input methods. An **InputGesture** object

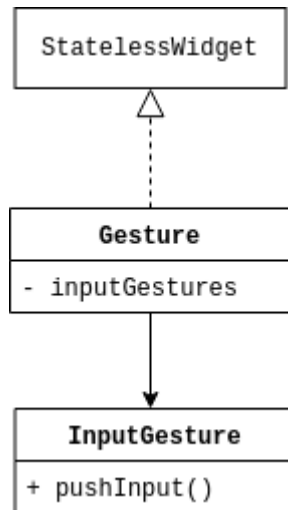


Figure 8: Gesture.

contains the **pushInput** method to communicate the inputs to the renderer (and thus the game logic). An **InputGesture** can be anything from a button to an area that detect a swipe and they must be defined by the designer of the game.

2.3.7 ScreenUtil

This class contains elements useful to deal with the screen orientation and scaling.

2.4 Game

This section contains a general description of the different classes implemented in the game specific module of the app.

2.4.1 InputGestures

The inputs gestures of the game all extend the aforementioned **InputGesture** class and the following ones were implemented in the game :

- A button for the basic attack.
- B button for the smash attack.

- Left and right buttons to move left or right.
- Jump button to (double) jump.
- Fireball button to launch a fireball.

The input gestures are instantiated using a factory pattern which makes it easy to create different sets of buttons.

2.4.2 SmashLikeLogic

This class implements the game logic of the game which is illustrated in figure 9. It basically :

1. Reads and applies the player inputs.
2. Fetches the data from the opponent using the *Bluetooth* communication.
3. Checks the hurtboxes to see if one fighter hit another and updates the damages and/or applies ejections accordingly.
4. Checks if one fighter is out of the border and decreases its lives counter accordingly.
5. Checks the end of game (one of the fighter doesn't have any lives left) and shows the end game screen accordingly.

Fetches data of the opponent include :

- Opponent inputs
- Opponent position
- Opponent damages and remaining lives.

This data makes it possible to keep both fighters synchronized.

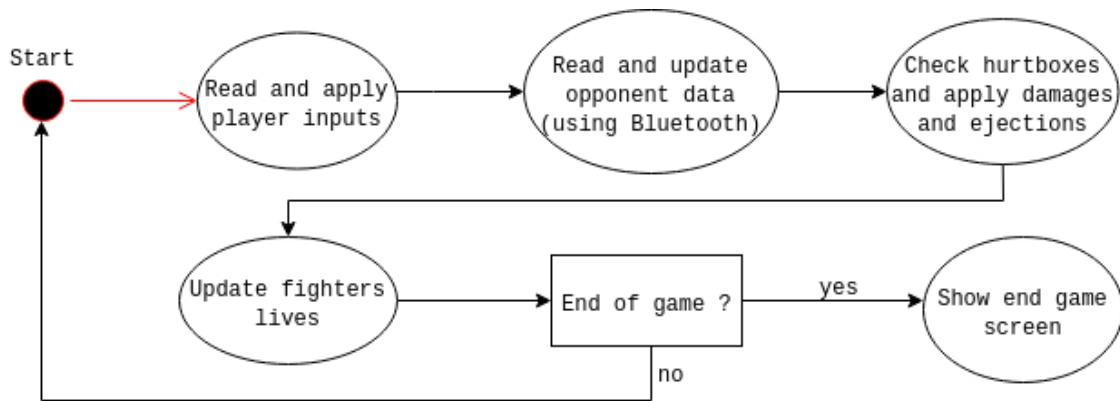


Figure 9: Game logic update function.

2.4.3 SmashLikeAssets

The **SmashLikeAssets** class extends the **GameAssets** class whose role is to hold all the assets of the game. It is built using a factory pattern with the **GameAssetsFactory** class, thus making it easy to build different **GameAssets** objects for different sets of maps and fighters.

There are three main categories of assets used by the factory :

- The UI assets.
- The fighter assets.
- The arena assets.

UI assets Those assets show useful information about the game state to the player. At this point, the UI is composed of the damage and life indicator of both fighters.

Fighter assets The fighters are physical assets that are controlled by the players. There is an abstract **Fighter** class which is implemented in the concrete **SantaClaus** class. There is also a **Fireball** class used to model the fireballs thrown by the fighters.

Arena assets The arena assets are all assets that compose the arena in which the fighters evolve. This includes :

- The background and cosmetic elements (non-physical assets).
- The platforms (physical assets).
- The invisible walls (physical assets).

2.4.4 Multiplayer

Two solutions were envisioned to implement the opponent. The first one was to design an AI and the second one was to oppose two human players using a *Bluetooth* communication, thus making the game multiplayer. The second solution was chosen because of the complexity to design a challenging and balanced AI though the multiplayer solution wasn't easy either.

The multiplayer is implemented in the **Multiplayer** class which uses the singleton pattern. Indeed, the **multiplayer** object is used in multiple places that are not linked each other (menus and game logic). Moreover, only one instance of this class should exist in the program.

A multiplayer game session works as follow:

1. One of the player is the server, he hosts a game session. The server decides the map and its fighter color.
2. The other player is the client, he joins a game session.
3. The server gives the map and color of its fighter to the client which is waiting for it. (**start** function in figure 10).
4. The game session is started and the players exchange their fighters data continuously.
5. Once the game is finished, the connection is closed.

The multiplayer uses the **Bluetooth** module as a mean of communication between the server and the client.

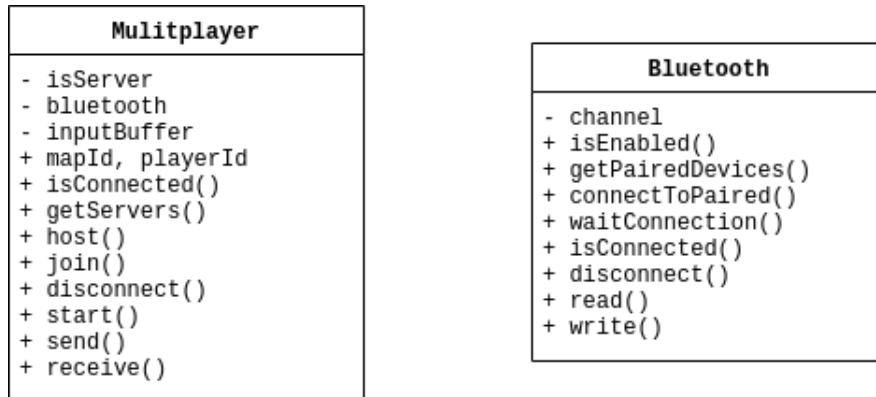


Figure 10: Multiplayer and Bluetooth.

2.4.5 Bluetooth

The **Bluetooth** module provides an easy way of managing *Bluetooth* connections between devices. It is implemented using the *Flutter*'s *platform channels* which is a mean of communication between a *Flutter* app (platform agnostic) and platform specific code that is, in this case, a *Bluetooth* handler module implemented on *Android*.

An overview of the exposed methods of the **Bluetooth** module is shown on figure 10.

2.5 Challenges

Two main challenges were encountered during the design of the game.

First, the **design of the game engine** required to provide an efficient, easy and reusable way of running a game which is not an easy task and many of its components needed a long reflection before being implemented.

The second biggest challenge was the implementation of the **multiplayer** with its *Bluetooth* connection. Indeed, *Flutter* offer no out of the box support for *Bluetooth* and this module has to be implemented from scratch using the platform channels. The good synchronization of the players while also keeping reasonable performances was also a challenging problem which is furthermore not fully resolved in the final implementation of the app.

3 Limitations

3.1 Problems

3.1.1 Bluetooth performance

One of the main issue with the *Bluetooth* module is that it must be run on the same thread that *Flutter*'s rendering thread. This is a limitation of *Flutter* which requires the methods of the *platform channels* used by the *Bluetooth* module to be run on its rendering thread. This leads performance drop since the I/O operations interfere with the rendering ones. There is thus a trade-off between the game smoothness and the latency of the multiplayer. The more the I/O operations, the less the multiplayer latency but also the less game smoothness and vice-versa.

The performances of the app from the submit version have been slightly improved by reducing the I/O operations to the benefit of the rendering.

3.1.2 Connection issues

After finishing a game, even if the connection between the two players is closed, the game was not responding if you tried to play another game and we had to completely close the game and re-open it in order to be able to play again without problem. There are also other connection problem that are more tricky to trigger.

3.1.3 Lives

We noticed that sometimes, not in a specific game state, when a player dies by being ejected beyond the game's edges, instead of loosing one life as it is suppose to be, the player may loose two of them.

3.2 Improvements

3.2.1 Optimization

Even if the performances were improved after the deadline, in order to have a more pleasing game to play, more investigations have to be made to improve the performances of the multiplayer with the ultimate goal of reaching 60 FPS.

3.2.2 Bugs

- The problem of re-connection triggered by the start of another game (see 3.1.1) has been fixed after the deadline which results in a far more appreciable game experience.
- Regarding the two lives loss issue (see 3.1.3), we noticed that it only occurs in multiplayer and not in the training mode, therefore, we suspect that this bug is due to an information transmission between the two players during a short de-synchronization, each player telling the other that a death occurred but at a slightly different moment.

The source of the problem might be different, but regardless, this issue can be fix with a bit more time of work on it.

3.2.3 Game

In his current form, *Santa Clash* is relatively good, however, there always are ways to improve and here is some ideas:

- Increasing the number of playable Santa (or even new characters).
- Building new maps.
- Allowing the players to choose the amount of lives they will be granted at the beginning of the game.
- Creating a *Solo* game mode where the player is against an AI.
- Adding music to the game.
- Creating maps with moving platforms.
- Adding a statistic page after a game to deliver information's to the players about the game they have played (e.g. time of the game, accuracy of projectile shots, damage dealt, etc.)

4 Conclusion

Overall, the project was fairly well conducted and the team managed to submit a usable app starting from scratch.

The game still lacks some content and main improvements could be made regarding the performances and stability (especially with the multiplayer) in addition to other bug fixes.

However, although it is not perfect, the app provides a fun and playable multiplayer smash-like game that also somehow represents a technical achievement considering the whole picture and the fact that *Flutter* is a rather new framework.