PewPewPew

Object Oriented Programming on Mobile Device

L'Hoest Julien s150703
Maréchal Grégory s150958
Minne Adrien s154340

Master Civil Engineer

Contents

1 Overview

1.1 Application Description

Our application, named PewPewPew, is a Worms-like game. It is composed of a home screen that can redirect, obviously, to the game but also to a simple tutorial and a shop.The tutorial is a simple sequence of images with a text, each of these explaining one step of the game flow and the associated controls.

In the shop, one can find different weapons that have different capabilities compared to the default weapon (which is a fist), such as knockback strength, the number of ammunition or the explosion radius. These weapons can't be bought with real money but are acquired thanks to the money earned at the end of each game (which depends on the number of kills, damage dealt and taken, ...). For the sake of testing, the current version of the application is embedded with 4 weapons available from the beginning.

The game in itself is a turn-by-turn multiplayer Worms-like game in which 2 to 4 players have 1 to 4 characters. The goal is to be the last player alive (i.e., with at least 1 non-dead character). A character can die either by taking too much damage or by being knocked out of the level[1].

The game flow is composed of the following steps : first, the first player chooses the character he wants to play by tapping on it. During this step, the player has the opportunity of observing the whole level by dragging the camera. After that, the character is given a certain amount of stamina with which it can either jump or walk. When the stamina is empty, or when the player decides it, a menu to select a weapon is shown and the player chooses its weapon (the menu can be closed and afterwards be re-opened if the player changes his mind). Finally, the player can aim and shoot with the selected weapon. The camera will then follow the projectile until nothing is moving anymore. At this moment, the next turn begins, and the next player can play.

1.2 Application Structure (Routing)

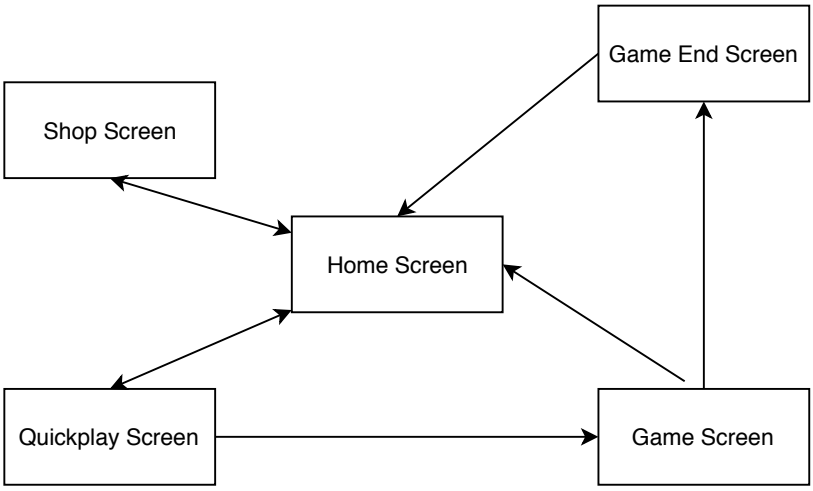


Figure 1: Application routing

The application structure is quite simple. It is represented on figure 1. The double arrow mean that we can navigate back to the previous screen with the device's back arrow. The user begin in the home screen, where he can either view the tutorial, go to the shop or prepare to launch a game in the quickplay screen. In this screen, he selects the number of players, their number of character and the level, before launching the game. The user can quit the game to go back to the home screen. But in order to free as much resources as possible for the game, we delete the old home screen when launching the game, so a new home screen is created when the user leaves the game. At the end of the game, the user is redirected to a end game screen where he sees his stats for the game and the corresponding amount of money earned, and can go back to the home screen.

2 Code Structure

The structure of our code is composed of three main sections which quite well represent the way we first divided the project. Each of these elements contains sub-elements that are described in more details below.

[1] Not to be confound with out of the screen, as the levels are bigger than the screen.

The first section of the code contains all of the menus of the application, their design, the navigation between them, the sounds and the handling of permanent storage for user-specific data. Its main sub-elements are the quickplay screen, the tutorial and the shop, the sound being handled thanks to the AudioPlayers package and the user-specific data thanks to the SharedPreferences package.

The second main element is the game logic, whose responsibility range from instantiating all the objects and loading the assets to responding to user events such as taps, long press and dragging, passing through the state management and the scheduling of the frames. Its main sub-elements are the GameMain widget and the GameState class, that use other elements, namely the camera, the entities and the weapons (with their projectiles).

The last element is the drawing system, whose responsibility is to fill the screen with the different elements, according to the current position of the camera, the positions of the moving entities and the predefined drawing order[2]. Its main sub-elements are the LevelPainter and the CustomDrawer (which is extensively extended) classes.

2.1 Menus

2.1.1 Tutorial

As stated in the overview of the project, the tutorial is composed of simple Text and Image combination to explain to the player how the game works. It is implemented with the showDialog() function of Flutter that open a new window. In this window, displayed a TutorialWidget that we defined, which simply display text and image, as well as arrow buttons to navigate through them.

2.1.2 Quick-Play

Before proceeding to the instantiation of a game, we need to ask the user his preferences about the number of players, the number of characters per team and the game terrain. We implemented that in a flutter bottom sheet which is triggered when the quick play button of the main menu is pressed. This sheet is composed of three different custom classes handling radio buttons, each one for each possible choice. The first two are classical groups of radio button but the last one has been modified in order to display images that are preview of the maps which are available. When the different parameters have been set, pressing the "Go" button will launch a game corresponding to these parameters.

2.1.3 Shop

When opening the shop, we first load the money the user currently has, and the list of weapons that the user can buy. This list is stored inside a json that is loaded when the user opens the shop. This json contains all the information about the weapons that need to be displayed in the shop, and that are needed for the game (its sprite, its price, the damage it does, what projectile it fires, ...). The user can then tap an item of the shop to buy it. If the user has enough money to buy it, an AlertDialog will be displayed to confirm the purchase. If the user cannot buy an item because he doesn't have enough money, or the item is already sold, we display an AlertDialog with the reason the item could not be purchased.

2.1.4 Sound

In order to implement a sound player in our application, we have use the "audioplayers 0.13.2" flutter package. It allows us to instantiate Audio Players instances and Audio Caches instances. The latter is a utility class to play audio asset of a flutter project by loading the data of the asset into a temporary folder to then be played. For time sensitive situation, it is possible to use a low latency mode of an Audio player and to pre-load an audio assets before using them.

Our application only need to loop over a background music and to play one or several sound effects in parallel during a game. So we wrote a custom class SoundPlayer which should be unique and accessible through the whole application, in order to be able to access sound functionalities anywhere in the code. We implemented this by using the "singleton" design patter, with a static field to store an instance of the SoundPlayer class, and a static function to access this instance. The main menu page can access this class to change the music or SFX volume, and each entity of a game can play a sound effect if it needs it. The SoundPlayer class in represented in the figure 2.

[2]Even if it is currently not that easy to use a non-default order, which consists in a "First-in First-drawn".

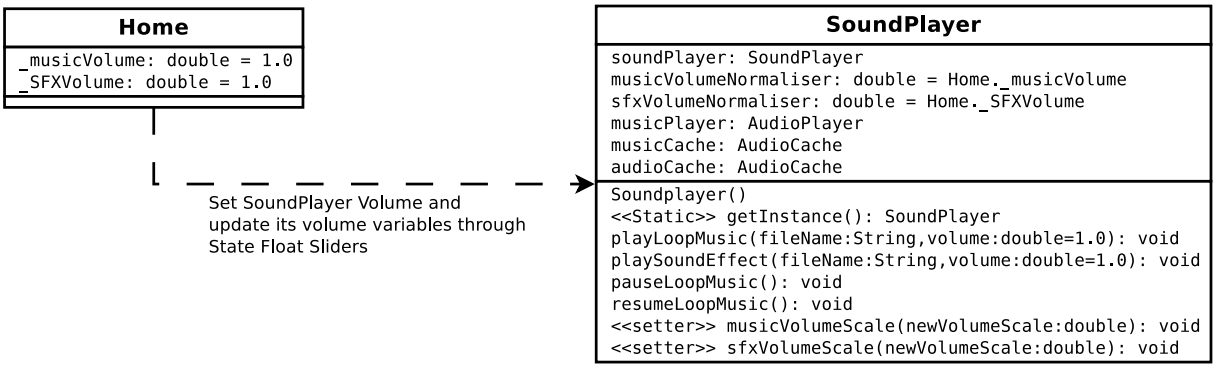


Figure 2: Sound Player

2.2 Game logic

2.2.1 Link between flutter and game and main game components

The structure of the game is displayed in figure 3. The class that is created by the quick play screen to begin a game is the GameMain widget. The widget will initialize the level, load the image assets and create the LevelPainter for the game, before creating a GameState class with the initialized fields. It is the GameState class that will contains all the game logic, without worrying about any of the flutter implementation details. The GameMain widget also implements a scheduleFrameCallback which is used to update the GameState at each frame. The GameMain widget is composed of a GestureDetector, that will forward all the touch events to the GameState class, and that contains a CustomPaint widget handled by the LevelPainter to display the game.

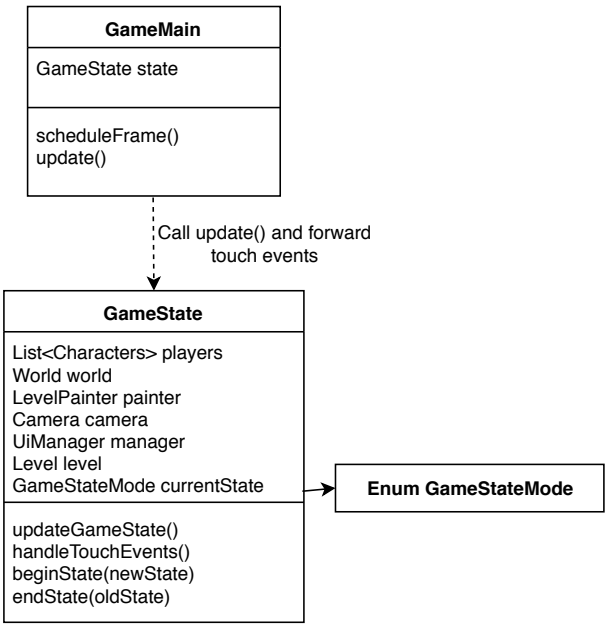


Figure 3: Game Structure

2.2.2 Game State management

The GameState class is the class that handle the game flow and manage the player actions. It contains a list of all the characters, a World class managing the game physics, a Level class describing the level, a Camera object and a manager for the the user interface (UI). The GameState update() function will be called at each frame by the GameMain to update the different elements of the game. It also contains functions to handle each type of touch event (touch up, long press, dragging movements).

The GameState most important field is probably the *currentState* field. The game is a turn by turn game, and each turn is divided into several phases. First player can move his character, then he can select his weapon, then aim, and then we show the results of the attack. We have clear delimitation between these phases, and in each

phase the action that the player can do and the things the game need to handle are different. We thus implemented a simple state machine in the GameState class, with each state represented by a value in the GameStateMode enum, and kept in the *currentState* field. We then implemented the beginState() and endState() methods to handle the actions needed to begin and end a state. finally, in the update() function, and in the function handling the touch events, we created switch statement for each state where we handle all the necessary actions.

2.2.3 Entities

When we were defining the structure of the game, we have defined that every object that exists within the context of the game (that excludes UIs) should have a visual representation on screen. And some of them should interact with the world physics. We have thus defined an abstract class Entity that contains a position, an hitbox, and a CustomDrawer object used to display the entity. If the entity should be drawn of screen, its drawer should be added to the LevelPainter. And if the entity should interact with other entities through the game physics engine, the entity should be added to the World class.

For entities which should be able to move, we specify a subclass of Entity which is called MovingEntity. They have additional methods and attributes which allow them to move in space through time. The class handling the game can then get access to the current positions, velocities and accelerations and applying the intern physic on these variables. Figure 4 is representing these two abstract classes.

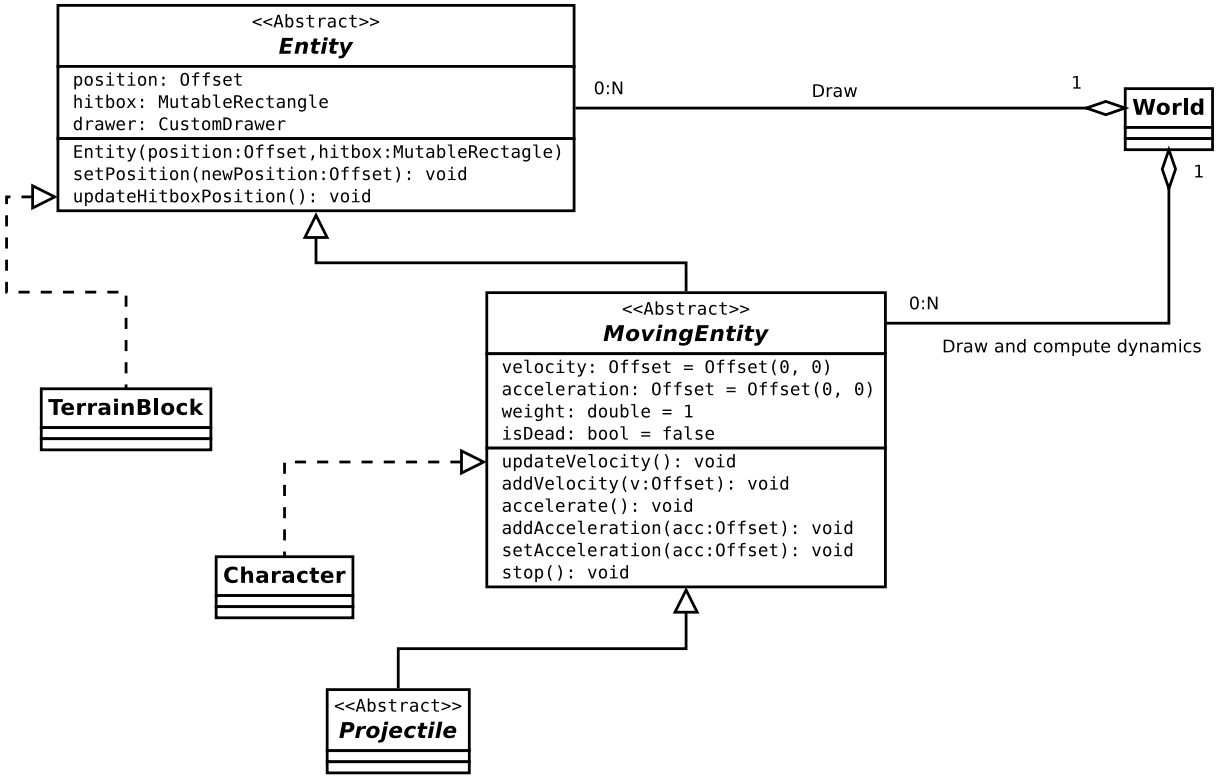


Figure 4: Entities

2.2.4 Physic engine

The management of collision and physic is always a difficult part of a game development. We limited ourselves to simple physics to not spend too much time on it. We thus decided that all the existing entities would be represented by a rectangle in the game physics, which make it easier to handle collisions. To manage this, we created the World class (5). This class is created and saved in the GameState. We can add entities to the world, and these entities will then be updated every frame with the physics we defined by the update() method that will be called by the GameState class at each frame. This update() method will apply the gravity to all the Characters and Projectiles, before moving them. After moving these entities, we check for collisions between them, and move back the entity if necessary, in order not allow an entity to pass trough terrain. Finally, we manage the explosions of the projectiles, if there is any projectile that is currently exploding.

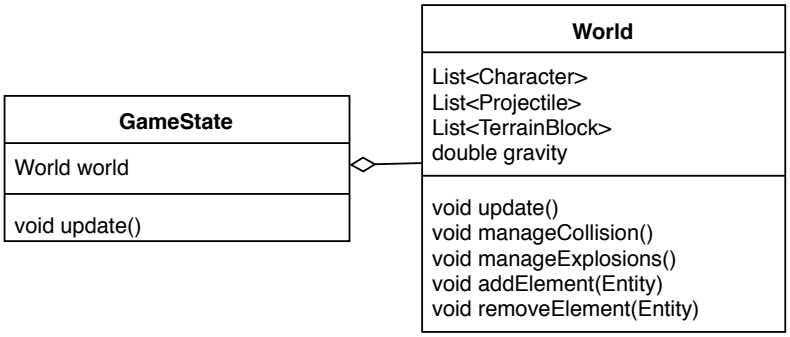


Figure 5: Physic and collision management in the World class

2.2.5 Level

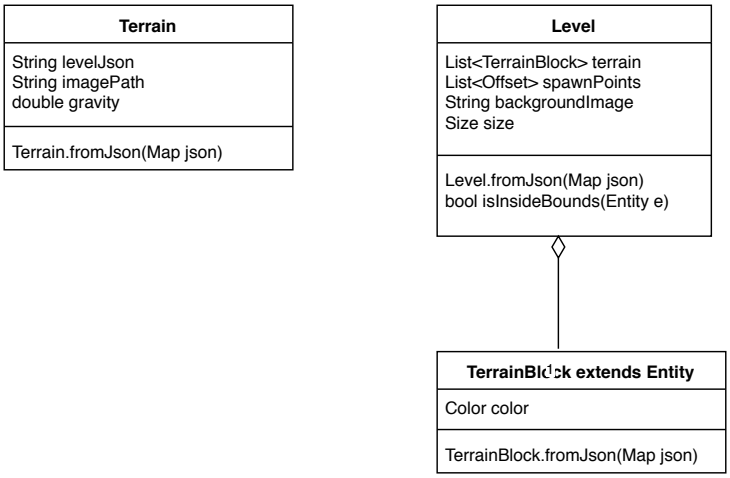


Figure 6: Level implementation

The architecture we used to manage levels is displayed on figure 6. We needed a way to save a level layout, and to get information about the available levels to show to the user in the selection of level. We thus created a json containing Terrain objects that will be loaded during the selection of level in the QuickPlay widget. These Terrain object contain the path to an image to display to the user to help him select a level, as well as a json describing a Level object. This json will be given as argument of the GameMain widget when launching the game.

The Level itself is composed of a list of TerrainBlock. A TerrainBlock is a simple entity with a rectangular hitbox, and a color to be filled with. The Level class contains also information about the size of the level, a list of Offset describing the possible positions in the level in which we can spawn a character, and some simple function to help detect when an entity goes out of bounds.

2.2.6 Character

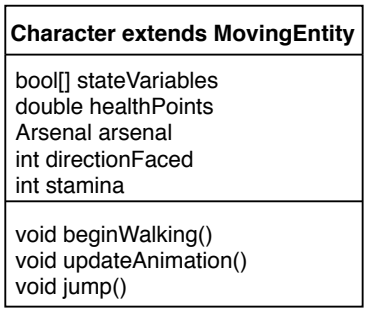


Figure 7: Representation of a character in the game

A character in the game in represented by an instance of the Character class (fig. 7) that extends the MovingEntity class. Each character have a health value and a stamina value, as well as an Arsenal class that is the list of all the weapons and their ammunition the character currently own. The character can walk or jump, which is implemented by the beginWalking() and jump() methods. The Character class also have a variable to represent the direction the character is facing (left or right), as well as a series of boolean representing the current state of the character (dead, moving, airborne, idle,...) to make it easier to draw and animate the character. The updateAnimation() method is a method that change the .gif file of the character depending on its state (to have different animations when the character is walking, idle, or jumping). This method should probably belong to the CharacterDrawer class, but we forgot to move it there.

2.2.7 Weapons and projectile

Weapons An important part of the game is the attack phase. When the current character run out of stamina or the player wants to attack, he has the possibility to select a weapon in its current arsenal. When a weapon has been chosen, the character will be able to throw a projectile which will damage characters within its range of impact.

Like the levels, we needed a way to store the characteristics of the weapons somewhere to be available for the game show as well as for the game itself. We thus created a json filled with a list of objects that contains all the different stats of the weapon, and this list is given as argument of the GameMain.

A weapon is a class representing characteristics and graphical asset of the weapon used. These characteristics consist in values of damages, knock-back, range, etc. A weapon can fire with the fireProjectile() method, that returns the list of projectile the weapon fire. These projectiles can have different variables (weight, initial speed, ...). The GameState method will call this fireProjectile() method when the player open fire, and add the projectiles to the World class. These projectiles are then handled by the World class like any MovingEntity. When a projectile explode of collide with another Entity, the World class will handle the damage done and the eventual explosions. It will then mark the projectile as dead with the isDead boolean, which will be detected by the GameMain that will remove the projectile from the game. The Figure 8 represents the relations between the projectiles, weapons and the main game loop.

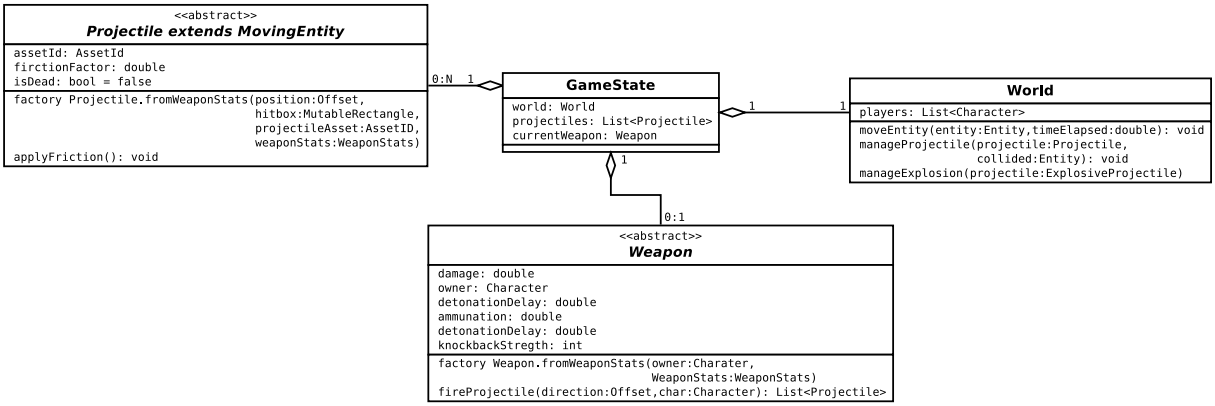


Figure 8: Weapon and Projectile in game context

Projectiles A projectile is defined by its graphical assets and different variables which will modify its behavior. Their are 3 different implementations of the Projectile abstract class. The first implementation is the Default-Projectile, which is a projectile that will deal damage to the first character it encounters. The second one is the ExplosiveProjectile, that will create an explosion either on impact, or after a determined time. The final implementation, which has no projectile implementing it at the time we submitted the project, is the Controllable class. This class allow the user to interact with the projectile through the GestureDetector. All of this is represented in Figure 9

Note that before the ExplosiveProjectile is removed by the GameState Class, the GameState class first call the returnAnimationInstance() method, that return an MyAnimation class. This class consist of a object with a special drawer displaying a single gif loop and play a sound alongside it. This is used to display things like explosions.

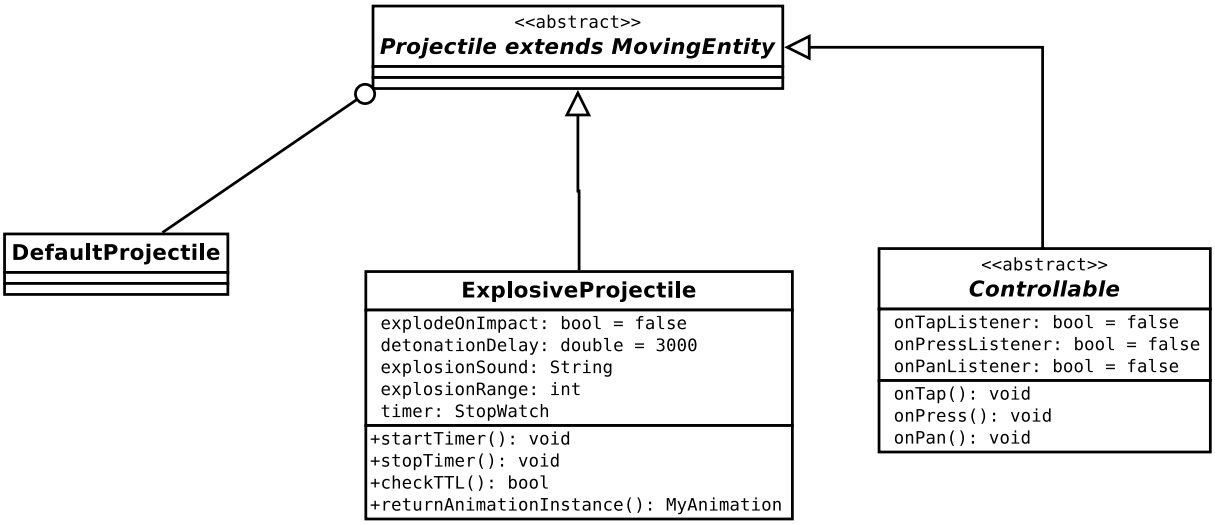


Figure 9: Different Projectile implementations

2.3 Drawing

2.3.1 Drawing system

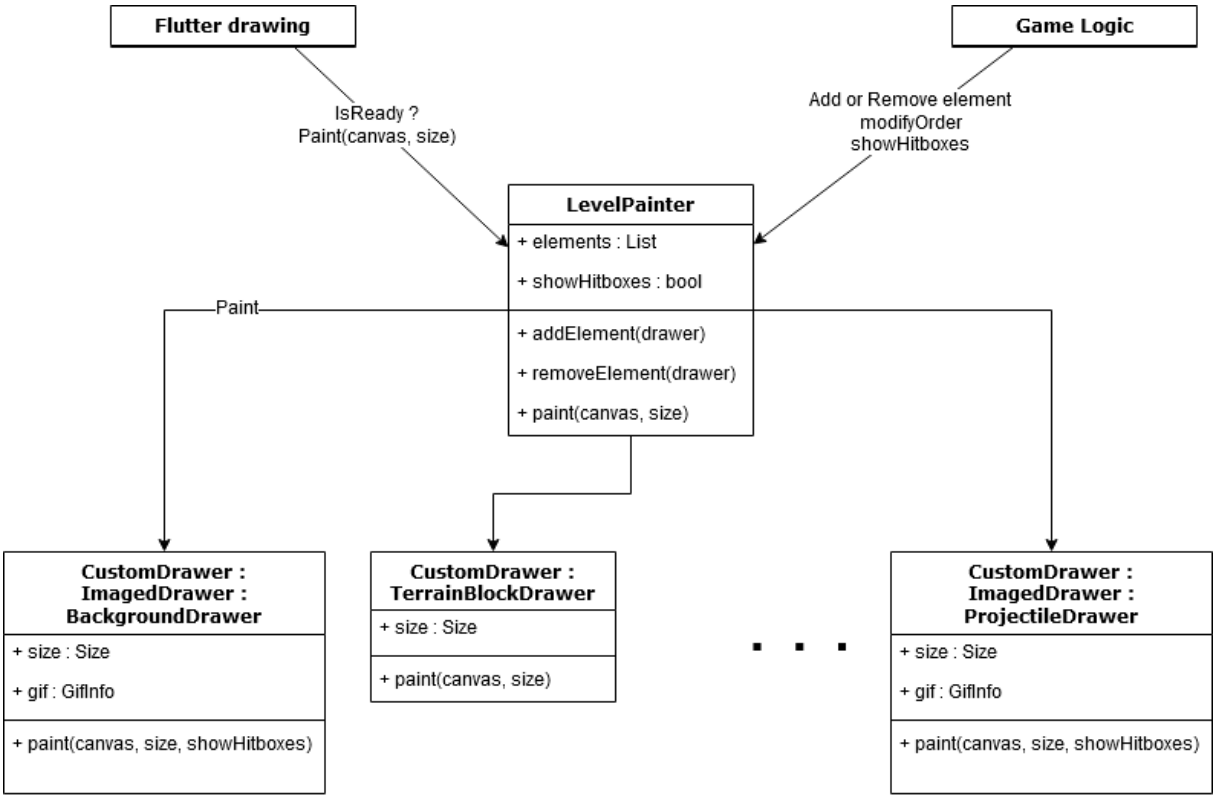


Figure 10: Summary of the drawing system

As we can see on Figure 10, the core of the drawing system is the LevelPainter. It has two main roles. The first one consist in acting as a supervisor for the CustomDrawer objects, telling them when to draw, possibly forwarding hyper-parameters, managing a loading screen at the beginning of the game, and handling the special-case situation in which a drawer would not be able to draw itself at a given moment (this situation should however be avoided at much at possible as it leads to a poorer user experience with elements that are replaced by "default values", or nothing). Its other role is to communicate with the flutter drawing system (i.e., responding to paint and isReady events) as well as with the game logic (removing or adding elements, modifying the drawing order, ...).

Conceptually, the CustomDrawers are simple elements whose only role is to paint themselves on the canvas, based on the current hyperparameters (in our case the showHitboxes boolean is the only one, which is used to display hitboxes to help debugging) and on the properties of the physical element they represent (position, state, size,...). More technically, we created two abstract classes which are the CustomDrawer and the ImageDrawer that contains a great part of the code used by the drawers. They are indeed responsible for managing the elements that almost all drawers need such as their size or their image (which often change as we use gifs files to represent animations). The children classes almost only override the paint function (and the constructor) except for the TerrainStrokeDrawer which implement a small algorithm to compute the set of strokes of a terrain, given its set of blocks (this is useful to not paint strokes between blocks that are side by side).

An element that is important to notice is that all the sizes that we use in the drawing system are relative, i.e. they are expressed in proportion of the screen height. Obviously, this is done in order for the system to be able to adapt its size based on the size of the user's phone, however another advantage is that the elements' proportions are kept constant even if the screen is not 16:9, which means that they are not (or should not be) ugly. The conversion is only done inside the paint functions, except for the images that are preloaded using the size obtained in the home screen (which we assume should be the same as the size during playing).

We use the AssetManager class to load (and pre-load before the game starts) images, and keep them easily accessible.

2.3.2 Camera

With this methodology for the painter we can simply implement the management of the camera by shifting Flutter's canvas by an offset. The Camera class contains thus the Offset of the current location of the camera, as well as a method to simply center the camera on a given character. It also implements a mechanism to manage inertia for the camera, to keep the camera moving for a bit after the player dragged the screen.

2.3.3 Weapon Selection and UI management

Weapon Selection The weapon selection interface is particular in the sense that we wanted to use something else that our usual rectangle+image combination. Indeed, this interface is drawn as a circular list around the character that contains a set of circular boxes to select the weapon (see screenshots in the Annex). This box, whose color is chosen accordingly to the character's team (or grey if the weapon can't be chosen because it has no ammunition left), contains an image of the weapon. It took us a little bit of math to ensure that the list radius is big enough to avoid overlaps and to compute the positions of each elements, but we are glad of the result which we find better than a set of rectangles randomly placed on the screen.

Other UI The rest of the UI is implemented in the UiManager class, where we implemented methods to easily add and remove UI components. For each component, we implemented a CustomDrawer that is initialized in the UiManager class. The UI components implemented are :

- The stamina bar to display the current stamina of the character
- A component to easily display text in the screen, and have it fade out after a pre-defined duration
- An arrow to indicate the direction of the character jump/its aiming with a weapon
- A marker to indicate the location the character is moving towards

3 What is missing

3.1 In the code

Globally our code implements all the needed functionalities for the game. The only missing functionality is that we currently cannot easily modify the order in which the drawers are painted. This means that it is painful to, for example, put the character currently selected on front of all the other characters. It is a relatively minor thing but that could be useful to have.

3.2 In the game

There are several ideas for the game that we couldn't implement due to lack of time, and because we already met the lines of code requirements for the project. These are :

- Implementing an AI to play alone. This could have been done with a rather simple AI, by simply targeting the closest enemy, or with a more complex AI with minimax algorithms.
- Implementing a destructible terrain. It would have been nice to handle destruction of terrain when a projectile explode. That would lead to more dynamic games with changing terrain.
- Implementing a level creator. Adding the possibility for the user to create its own level would have been nice for the the replayability of the game.
- Online mode. Since it's a multiplayer game, the possibility to play online would be a plus. However, it would require a lot of efforts to make it work.

The game could also be improved without any new functionalities, with just more content. We could add new weapons, new levels, new looks for the characters, ...

4 Problems

Currently, the only issue we noticed in the application are in the sound management. One issue is that we don't use any audio focus, so the game sound is playing over the other sound of the device. This is not implemented in the audio package we use, we would need another package or implementing it in Android. The other issue is that the sound of the game keeps playing when the user switch the foreground app on the device.

5 Technical Challenge

The main technical challenge that we faced was to understand and interact well with Flutter's framework. The other aspects, for instance the maths for handling the elements' position and the game state, were not too hard as we decided to keep things simple with rectangles (and circles). For instance, at the beginning of the project, we wanted to have several painters in order to only redraw elements that have changed (using the isReady callback). We had a hard time trying to do so before finally understanding that the framework does not work this way and erase everything contained in the painter when rebuilding. Another example is the DrawImage function of the ImagedDrawer, which enables to rotate, flip and resize the image being drawn. It was once again quite challenging to really master the canvas to make it do what we want.

6 Run the code

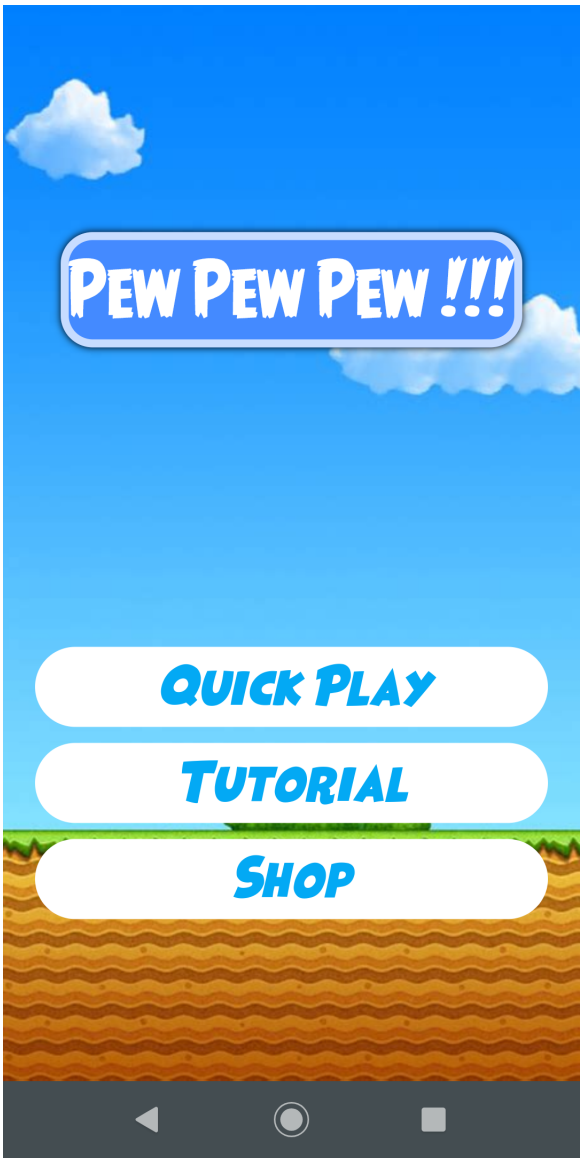
The code is a standard flutter project using Gradle, so you need to import the Gradle project to run the code. No particular setup is required except to ensure that you have the good versions of dart and flutter as well as a SDK version high enough. The following versions work on our machines :

- Flutter version 1.10.7
- Dart version 2.6.0
- Android SDK 28 (but 16 should be enough according to the build.gradle targetSdkVersion)

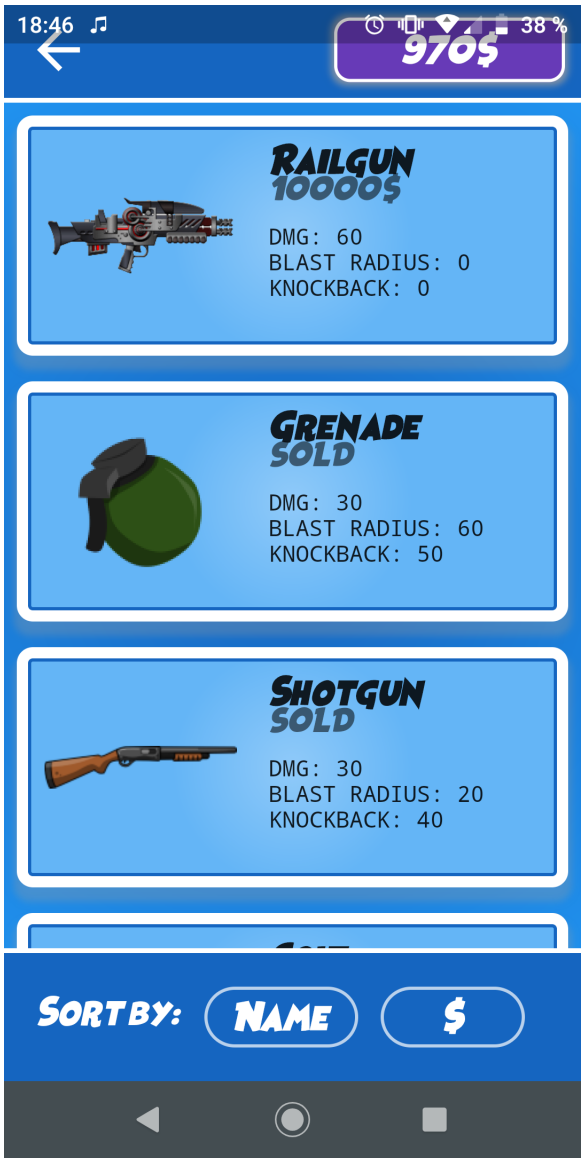
With this configuration, there is a well know issue when compiling stating : "More than one file was found with OS independent path 'META-INF/proguard/androidx-annotations.pro". It is recommended to upgrade or to add an exception within the android/app/src/build.gradle in the android section:

```
android{
    ...
    packagingOptions {
        exclude 'META-INF/proguard/androidx-annotations.pro'
    }
    ...
}
```

7 Annex : Some Screenshots



(a) Main menu



(b) Shop of the game

Figure 12: Weapon selection in the level