Object-Oriented Programming on Mobile Devices

Sceptical Go

2nd MASTER CE

Emeline Marechal
s142621

Sami Ben Mariem
s140915

Natan Derroitte
s141770

December 14, 2018

1 Application description and structure

Sceptical Go is a 2D puzzle game where you need to find your way in space among multiple planets and asteroids to reach the end of the level!

Your only mean of propulsion is to use the gravity of the planets around you to take the right turn at the right time. If you miss your shoot, you will be lost in space forever, wandering indefinitely...

Fortunately, you can place planets with different properties, as well as portals and other stars on your path to help and guide you towards the end of the level.

Find the right combination of planets to deviate your course and form a path in the big empty space so that you can land safely in your shelter! But pay attention to passing asteroids and other obstacles! You could crash into them and loose your spaceship!

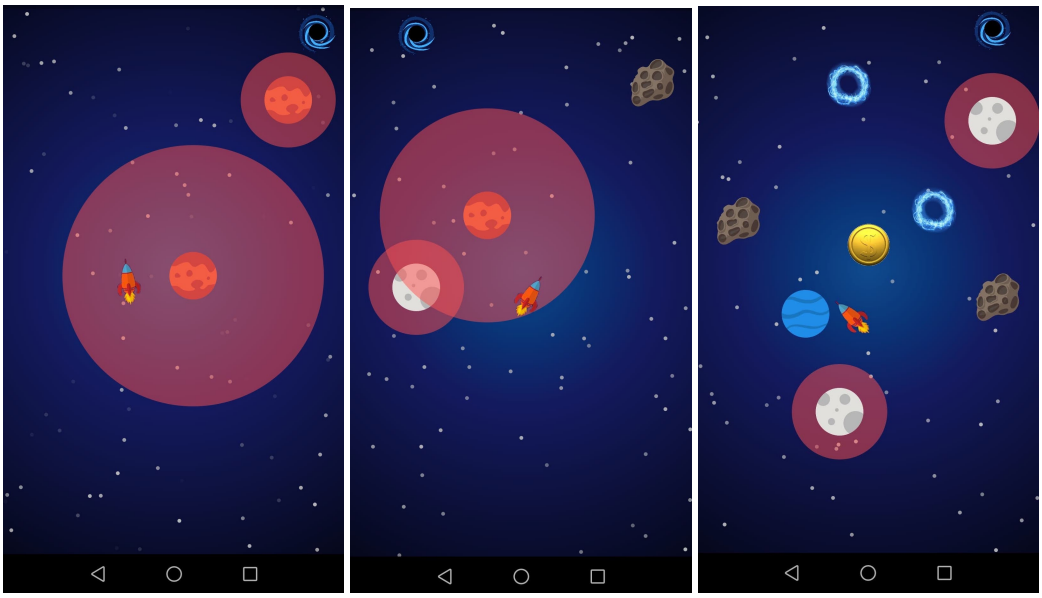

Figure 1: Sceptical Go

How to play

For each level of *Sceptical Go*, the player has to bring the spaceship safely to the other end of the screen, while avoiding different fixed obstacles laid out before them. Additionally, the player has at their disposition a set of different *actions* that they can drag and drop on the screen wherever they want.

The goal is to find the right combination of planets, portals, etc, in order to form a path towards the shelter. The different actions available are:

- Planet: The planet is characterized by a gravity field. If the spaceship enters that zone, it is deviated more or less strongly. This gameplay element can be used to deviate the course of the spaceship.

- Sponge planet: The sponge planet has no gravity and doesn't attract the spaceship. Rather, when the spaceship bumps into a sponge planet, its course is deviated with a rebound.
- Portal: The portal can be used to travel across the screen from one point to another and to avoid obstacles.

Routing

The routing of the game is quite simple: the first screen is a welcome screen, that will take the player directly to the Galaxy Menu. Once there, they can go into galaxies that are already unlocked. They arrive then into the Level Selection Menu where they can choose to play certain levels.

2 Technical aspects - Code Structure

Our project structure is organized into four folders:

- Menus: The store to choose one's spaceship, as well as everything related to menus and navigation between them.
- Helper: Different helper files to deal with layout on screen, sharedPreferences, etc.
- Onboarding: Everything related to the onboarding of the user, as well as several hints displayed as new gameplay elements are unlocked.
- Game: The core of the application, where the game and its logic are actually implemented.

In the next sections, we will review the application workflow, the pattern used for the game logic, how the collision detection was implemented; as well as other elements that constitute the application.

2.1 General code structure for the Game part

Sceptical Go is a puzzle game organized in different levels, where each level is described by a file in the AssetBundle. This description includes the positions of static planets that the player can not move, as well as the planets that are available to them to place on the screen and create a path.

When a user selects a puzzle, a new instance of GamePlay is created. This Widget is the entry point into the game strictly speaking. GamePlay is responsible for loading the puzzle from the AssetBundle and create an instance of Puzzle object. The Puzzle object holds all information about the current state of the puzzle: what are the positions, velocities, etc, of all game entities.

Once the loading of the puzzle is done, the GamePlay instance will call layout_puzzle on the Puzzle instance to recompute game entities positions based on the screen width and height. Indeed, entity positions in the AssetBundle file are described for a certain screen size. Therefore, it is necessary to recompute every positions as well as image sizes in order to keep the correct ratio, and for the level to stay exactly the same on all devices. Otherwise, the solution to the level might change, and the level could even become impossible to solve, as everything is based on entities positions.

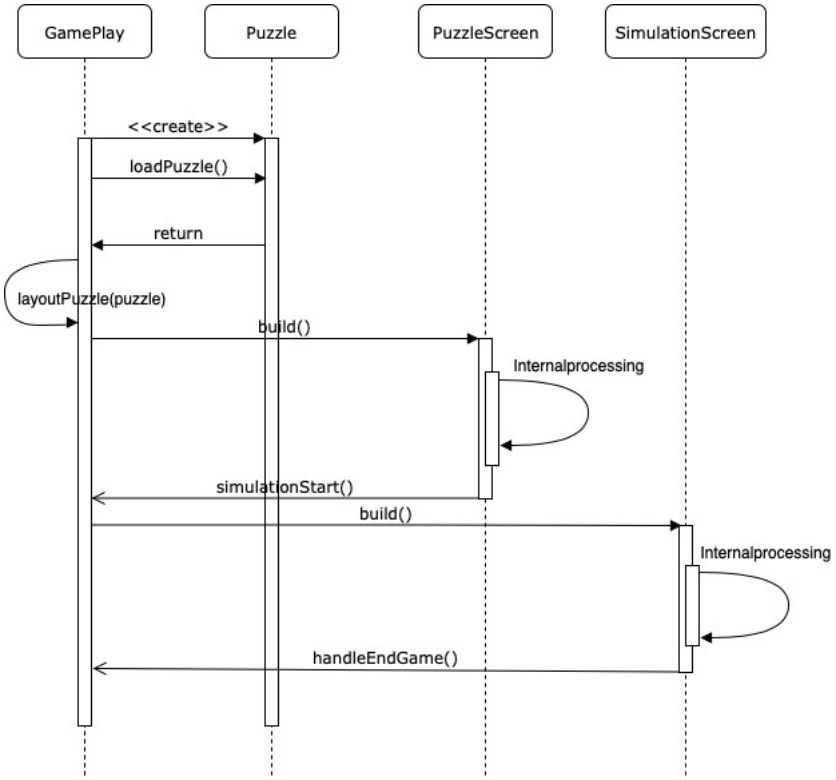

Figure 2: Gameplay workflow

After this loading phase, the player can start playing the game. From there, the application workflow is subdivided into two main parts, that we call *puzzle phase* and *simulation phase*. Indeed, *Sceptical Go* is first and foremost a puzzle game, where the player has to lay different elements on the screen to find a correct path. This corresponds to the *puzzle phase*, in which the user can drag and drop planets on the screen.
After the player is satisfied with their action, they can enter the *simulation phase* and launch the spaceship to see if their solution is correct and if the spaceship reaches the shelter. This is an animation phase with no user input where the trajectory of the spaceship will be displayed. The real-time position of the objects on the screen will be computed mainly using real-life physics : Newton's law of universal gravitation and Newton's second law.

These two phases are managed by the GameEntity Widget, that will either return a PuzzleScreen Widget or a SimulationScreen Widget. The navigation between both screens are managed by different callbacks. Once the game is over (the player either won or lost), GameEntity will take care of either launching the next puzzle, or restart the current puzzle.

You can find on Figure 2 an activity diagram summarizing these steps. Additionally, you can find on Figure 3 the UML diagram for those classes.

2.2 Entity-Component-System Pattern (ECS)

The game logic in itself, that is to say the animation of the different game entities, their behaviour, their collision, etc, we chose to use the ECS pattern, which is quite common in game development.

The idea of this pattern is to use *composition* over *inheritance* to define game entities, which allows for greater flexibility. More precisely, a game entity will be reduced to a container of

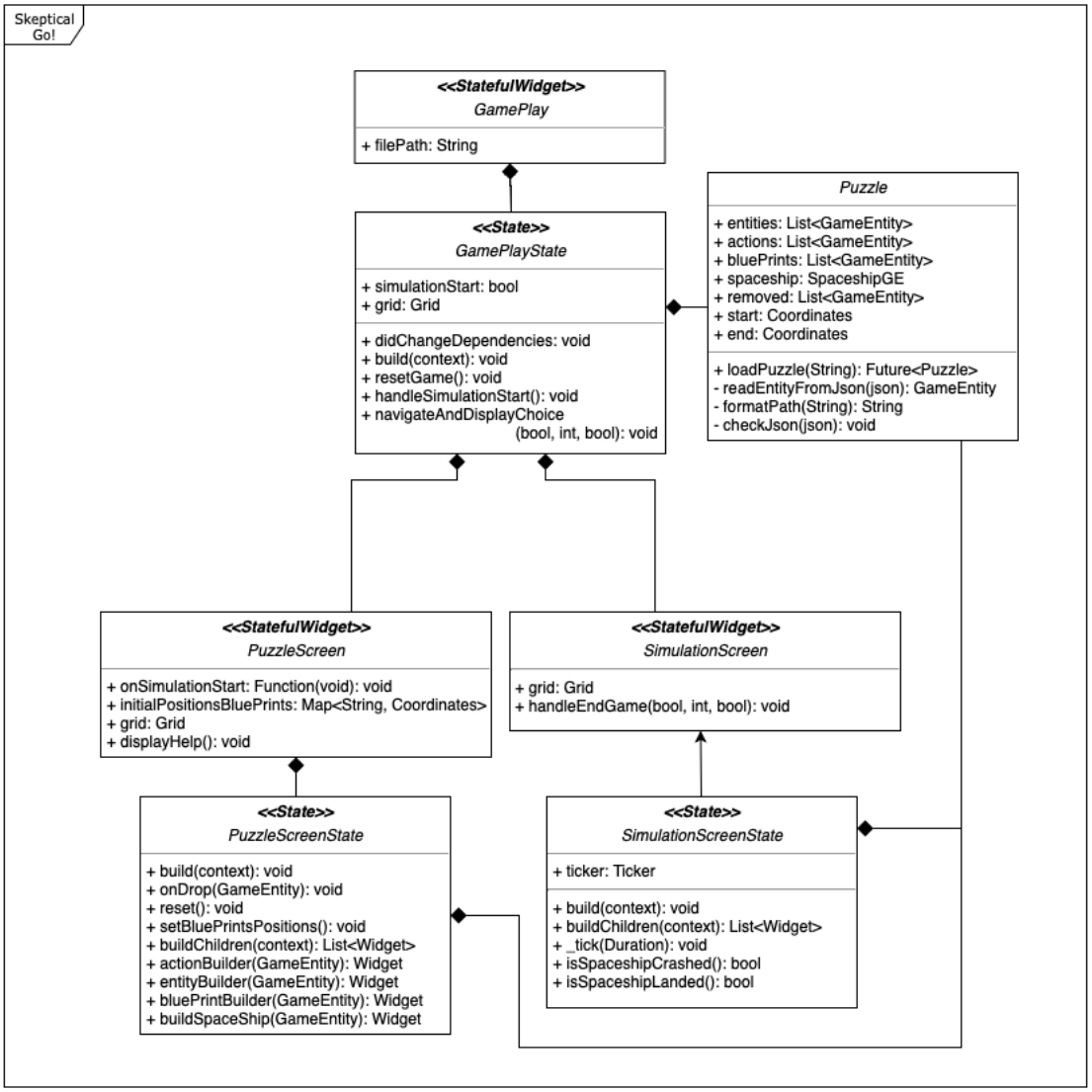

Figure 3: Game Phases

components, that one can add or remove to/from the game entity in order to add behaviour and functionality to the entity.
That way, it is possible to define entities and their behaviours simply by picking whatever components needed.

This pattern solves the problem of complex hierarchies that are difficult to maintain as new elements are added to code. It also allows to keep different domains (graphics, collisions, etc) well separated from each other.

You can find on Figure 4 the UML diagram for this pattern applied to our game logic.

2.3 Menus

The creation of menus consists in a set of Widgets linked together in order to be able to navigate in the application in an intuitive way.
The main challenge regarding the menus was to keep the player's progress (in terms of levels and pieces collected), up to date as they progressed. It was indeed necessary to update the user's coin number when he/she goes back after having brilliantly won a coin in a level.

To do this, two mechanisms have been put in place. The first was the communication between widgets via a *StreamController*. This is what is used in particular to update the position of galaxies when the user drags the Galaxy Menu.

The other mechanism was to carefully update the changed values using the *SharedPreference* package and read the corresponding value when *re*-building the menus.

Apart from these communications, the implementation of the menus themselves is simply a matter of manipulating Widgets so that the visual result is appropriate to what we wanted.

2.4 Onboarding

To improve the user experience with our application, and guide them in how to use the game, we provide the user with a small tutorial, as well as little hints when new gameplay elements are unlocked.

More precisely, a main tutorial is provided the first time the user launches the application, explaining how to play and what are the main objectives of *Sceptical Go*. Additionally, when new elements are introduced in the game, a small tutorial is provided to explain it. At any time, the user can also press a long moment on different elements for that same tutorial to be displayed again.

We use the package sharedPreferences to remember which tutorial the user has already seen and not repeat ourselves. We also use the package video_player to display the tutorial videos. Additionally, we would like to mention that the code for the layout of the main tutorial (swiping pages and background) has been strongly inspired from here.

2.5 Collision Detection

For the game to work, a collision detection system is needed. Indeed, when two entities collide, a specific action needs to be performed by one or both of the entities concerned. This section will describe how the collision detection is handled.

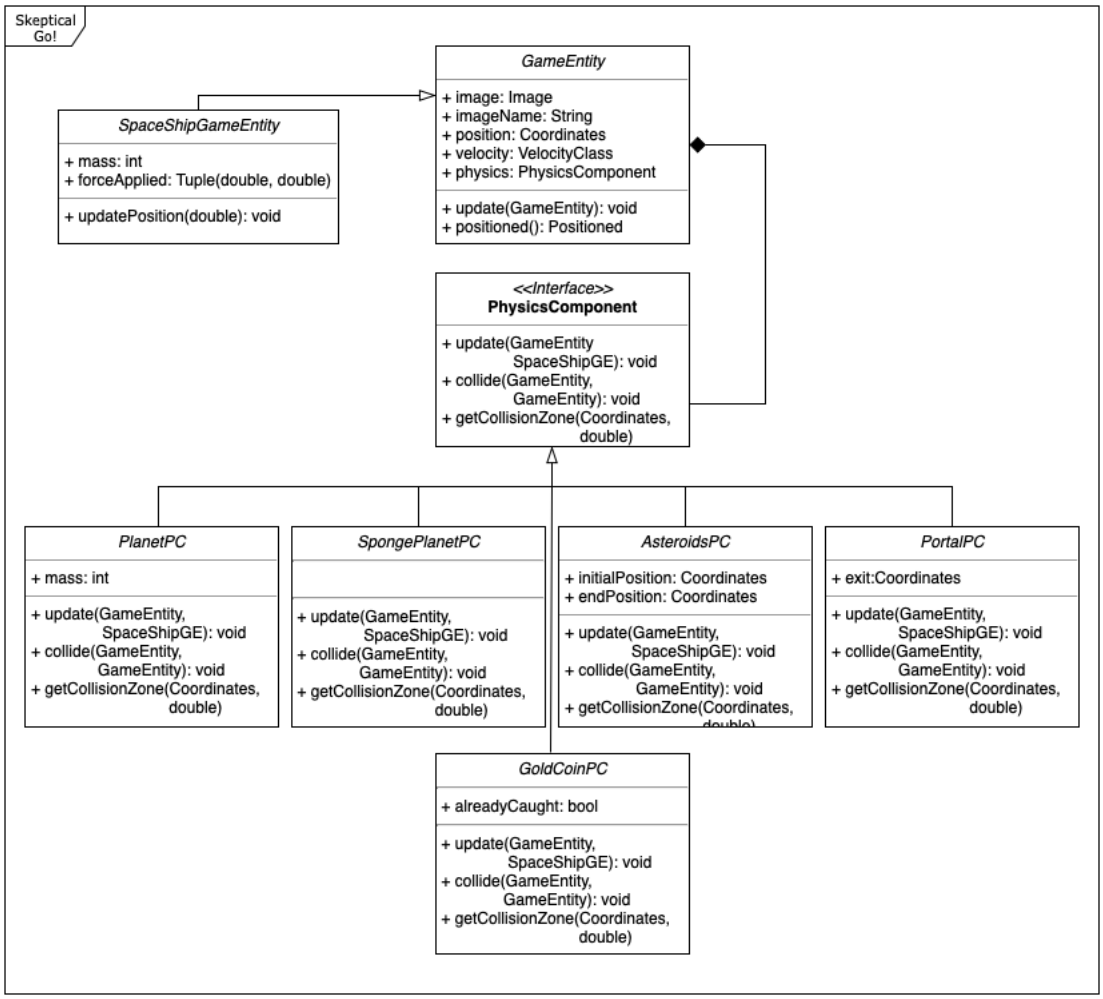

Figure 4: ECS pattern

Brute-Force

The first and naive idea would be to simply check for every entity if it is colliding with all the other entities being considered. Unfortunately, as we have several moving entities, this option would be inappropriate and inefficient as the number of comparisons will be very high (check for intersection of pixels positions).

The grid

To solve the problem, a grid has been used. The collision detection system consists in recording the position of all the entities in a grid. Therefore, moving an entity in the grid will simply consist in updating the part of the grid related to this entity and to check if the cells where the entity is moving to is not already occupied by another entity.
The grid implementation is working the following way :

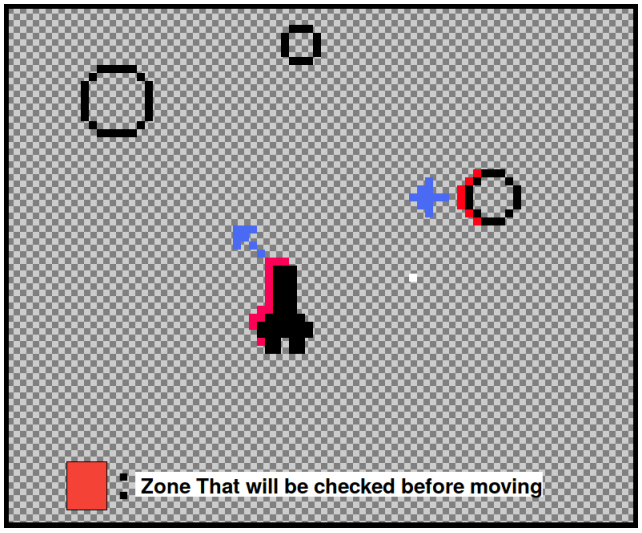

Figure 5: Grid System for Collision Detection

In order to get collision zones as close as possible to the images that we took, we decided to use a grid whose cells represent a single pixel of the real image.
Then, we will initialize the grid with all the *initial positions* of the entities. Finally, when an entity wants to move, it will update its position in the grid. This technique is the perfect illustration of the trade-off that will prefer to increase the space-complexity in order to get a better time-complexity.

3 Improvements

In this section we will review the limitations of our Flutter application, as well as different elements that could have been implemented with more time.

Asteroid animation

A downside to our application is the asteroids animation. Indeed, the animation only starts when the player launches the spaceship. During the drag and drop phase, the asteroid stays immobile,

making the game a little bit awkward at times.
This is a consequence of our initial game architecture into two phases: the *puzzle phase* and the *animation phase*. Indeed, during the *puzzle phase*, no animation is implemented, only a drag and drop system. Only the *animation phase* has a main loop to make elements move on the screen.
As implementing asteroids animation would have require a big change in our architecture, and as this problem doesn't prevent the game from working properly, we decided to leave things that way.

Spaceship launcher

The launcher for the spaceship is sometimes a bit reluctant. The key in using it is to make a long smooth gesture in a straight line. With a little bit experience, it becomes easier to use it. However, for the first-time comer, it may look like the game is a little bit buggy. You can find the reason on Figure 6.
With more time, we could have implemented another type of launcher à la *Angry Birds*, to gain precision when shooting the spaceship.

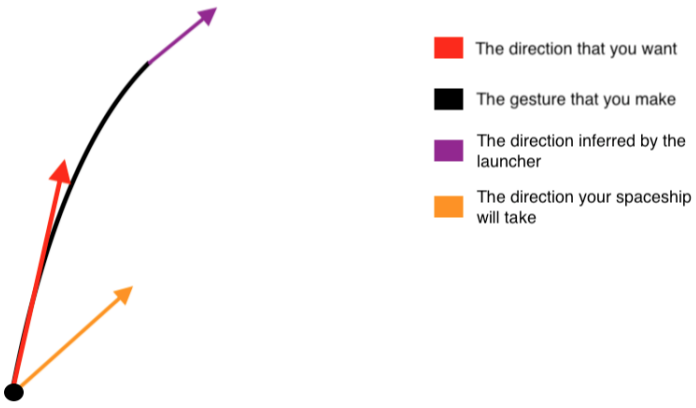

Figure 6: Launcher

Sponge planets collisions

A shortcut that was taken when designing the sponge planets was also to limit bounces. Indeed, depending on the relative position of the object that collides with the sponge planet, only four outcomes are possible which consists simply in reversing the horizontal or vertical velocity of the object.
An improvement in the physics of the game, by encoding elastic collisions between objects, could have been achieved. However, after testing our simplified model, the solution created seemed very relevant to us: the rebounds are easier to understand with this simplification and makes the game easier to play.

Gameplay features

It is always possible to add more gameplay elements to make the game funnier and enjoyable. However, we were limited in time and we restricted ourselves to the set of basic features to have a working application. Even though, we thought of some additions that would have make for a more complete and professional looking game.

Currently, the application introduces three new game components as the levels progress: Sponge Planets, Asteroids and Portals. It would have been possible to create other features, such as aliens or people that you have to rescue, forcing you to take a certain path over another.
In the same vein, we could also have created more levels, increasing the game's lifespan.

Moreover, several other features came to mind during the development of the application. Let us note for example the addition of music accompanying the player in their adventures or the implementation of purchasable bonuses allowing the user to further customize the puzzle game.

4 Problems

When building the apk for the Flutter Application and trying it out, we noticed the application was lagging after having displayed a video (main tutorial or hint during the game).
As this problem is not present when running the application with Android Studio/Visual Code, we do not know where the lag comes from, and we were not able to solve it. To display videos, we used the package video_player, which is a plugin developed both by the Flutter team and by the community. Therefore, we assumed that the problem comes from this plugin, as it still under development.

5 Conclusion

At the end of this project, we created a mobile game with a gameplay inspired by Newton's law of universal gravitation.

Concerning the game itself, it has been divided into two parts. The first one, the *puzzle phase*, concerns the puzzle where the player can place their actions; and a second one, the *simulation phase*, where the movements and collisions of objects are performed using the Entity-Component-System Pattern. In addition, the navigation in our application is based on a Menu system that we hope is intuitive.

By nature, our application leaves room for improvement, but we are satisfied with the work done. The application created seemed fun and pleasant to play to us.

Beyond the product, this project allowed us to familiarize ourselves with mobile development and in particular with the Flutter environment which can only be positive for students in our field.