1. General Information

The project is a match3 game, very similar to what Candy Crush or other match3 games do. It is a simpler version, as the scope and time were limited.

Before getting to the technical points, please disregard any visual non-matching color choices of the game (in general). I'm neither a good artist nor have a nice common sense for color matching and choosing. I did my best to have a coherent visual game.

The following requirements needed to be met:

- Grid containing pieces with different colors. [DONE]
- The player can switch adjacent game pieces. [DONE]
    - (Allow both dragging or tapping.) [DONE]
- Upon executing a switch, if it leads to a match of three or more pieces of the same color in either a row or a column, those pieces should be removed from the game board. [DONE]
- After pieces are removed, the game board will collapse. This means that pieces above the removed ones will fall down to occupy the empty spaces while new pieces will fall from the top to fill the board. [DONE]
- Score system where every match gives a score [DONE]
- Time-based score game mode; each game has a specific amount of time to achieve the highest possible score [DONE]
- Game loop [DONE]
    - Start screen
    - Gameplay
    - Result screen with score

As referred here, all mandatory requirements were met. Some non-mandatory requirements where added, as it was stated that a good user experience was very important, I added some extra requirements I found important in order to achieve this:

- Background music (one for the menu and other for the game scene) were added to make the experience more amusing.
- Sfx were added for different types of matches and wrong movements.
- Sound settings menu, to control the volume of both background and sfx sounds (in both main menu and game scenes).
- Visual countdown with numbers with an animation when time is almost over. In addition a regressive time bar.
- Pause button to stop the countdown in-game.
- If the game is unfocused or minimized (on android/ios), the timer is automatically stopped and resumed when focused again.
- An automatic mode. This addition was just for fun, there is an "automatic" play mode where a bot plays the whole game (or until pressed the button again to stop it).
- System to reshuffle the board in case no matches are possible.
- In general, every parameter to configure the game is available through the editor.

2. Project structure and architecture

The project is composed of about 20 scripts, where the majority of them just control specific UI elements.

1. Core Systems

The core of the game is in the Game scene. This is separated into 3 main systems, and also the scripts for the visual elements. Those systems are:

- GridSystem: This system is the heart of the project. It is separated into the core engine of the match3 and the connection to the visual and input part.

  The core engine is agnostic of the Unity Engine (except it returns Vector2Int, which could be replaced by custom classes). It is totally decoupled from the view/Unity UI too. This was made like this for 2 main reasons:
  - To be able to reuse the system in other projects, even if Unity is not used.
  - To be able to test the core system of the project in Unit Tests, without worrying of Unity visual elements or other problems non related to the board mechanics. Although no Unit Tests are written for this project, the code is prepared for it, the exposed API of this core engine is ready to be used in this kind of tests.

  There are some particularities about the implementation, but one of the more remarkables is the shuffling if there are no possible matches. I don't know if this was part of the requirements, but I implemented it. If for some reason, at some point, no matches are available on the board, the system detects it and shuffles the board. It uses the Fisher-Yates algorithm, which I found works really well for this kind of situation. This algorithm does not guarantee 100% that a new matchable board will be shuffled, so if after N tries it cannot generate a matchable one, it generates a completely new board.

  Controller is implemented in the Orbe script, which simply calculates the direction of the swap of the finger and then it notifies the GridSystem that a swap happened.

  The implementation of GridCore has a 2D array of orbs to represent the state of the board. You will see that GridSystem has a Dictionary containing similar data, which represents the visual state of the board. This is because the GridCore is totally decoupled from the UI and Unity, and this link Dict ⇔2D needs to be done. The 2D array of GridCore feeds the Dict of the View.

- TimerSystem: This system controls the time. This time is established depending on the game mode. When the application is unfocused or minimized on a mobile system, the timer stops until the game is back.

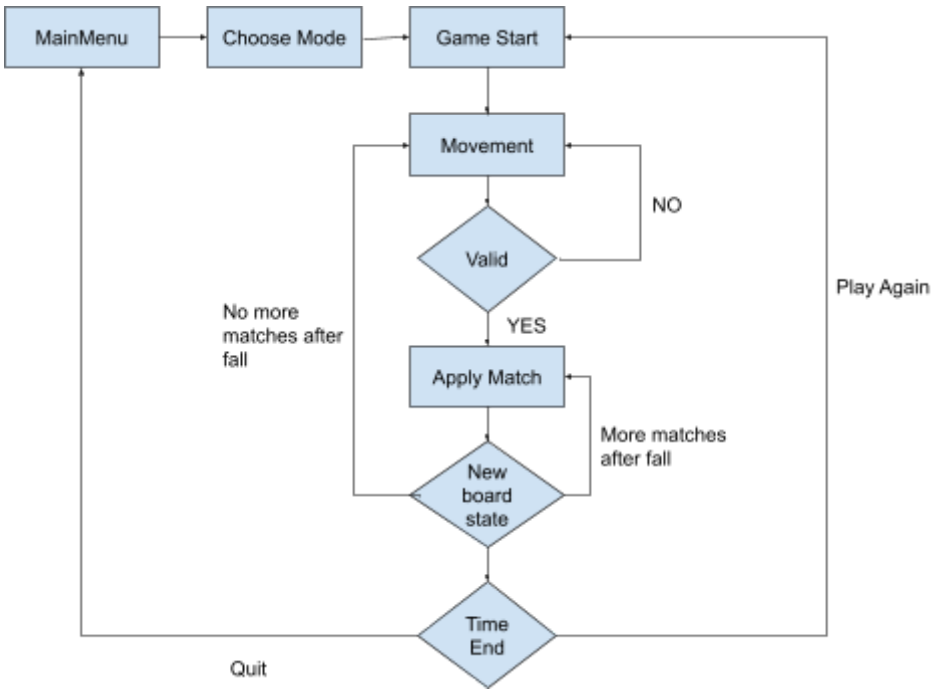
  There is also a pause menu, which stops the timer until it is closed.

  If time is over, the game is over and a panel with the score and other game statistics is shown.

- ScoreSystem: This system is in charge of getting how many points the player gets and also other statistics, like how many orbs of each type the player has matched. It also has a cool animation when a match has been made.
- SoundSystem: This system is in charge of playing music and sfx. There is one AudioSource for music and another one for Sfx. The music one plays in the background and can play one sound per time. The Sfx can play many sounds at the same time. There is a small limit system, because I want to be able to play the match3, match4 and match5 at the same time, but if 3 matches of the same type (match3 for example) is achieved, then 3 sounds of match3 are played at the same time and it is extremely loud, so this system limits how many sounds of the same type can be played in a same swap movement match.

The main menu is quite simple. There is a drop-down menu when Play is clicked showing the different timed game modes. This amount of modes can be modified through the editor (more on this later). There is also a settings button to change the sfx and music volumes.

I tried to make the code as straightforward and simple as possible, and it follows this simple state diagram:



2. Pool system

In some cases on the project, many resources appear and disappear during the gameplay, like the fruits or the texts for the score.

A very simple and generic Pool system is used to hold some instances of any element that need to be reused, so no Instantiating and Destroying is done constantly, as this could be a penalty on performance.

3. Tween system

A super simple tween system has been implemented for basic animations, like scaling texts (time getting over, scoring points…) or making the orbs disappear or fall down.

4. DontDestroyOnLoad

There are 2 elements that are on DontDestroyOnLoad. First one is the ConfigHolder, which contains some of the configurations of the game, but not all of them. The other element is the SoundSystem, which is used on both MainMenu and Game scenes.

Access to these two elements is done by “finding” them, and always using the same code in any place where this is needed. This is done like this so if the access mode is changed at any moment in the future, a simple search through the whole project with an IDE will get me the exact points where modifications need to be changed. This could also be achieved by using Singletons, but this other way was preferred, but any of them would work fine.

5. Bot

This was a completely optional feature and made it for fun, as the exposed API of the GridSystem and GridCore made it super easy to implement a bot that matches automatically until the time is over.

6. Not implemented

This is a list of not implemented features because of time and scope constraints, but that could fit perfectly on the project:

- Unit Tests
- Async loading of the scenes and resources (I didn't do it as the game is so simple that runs and loads fast on many devices)
- Improve visuals of the game in general
- Make possible matches to jiggle if the player doesn't do any movements (to help finding a possible match).
- Snapshot system, to be able to rewind to previous states of the board.

3. Configurations

Many parts of the project are configurable through the editor, let’s take a look:

- MainMenu
    - Number of game modes: This can be changed by adding more “Time Buttons” to the MainMenu Script attached to the canvas. More buttons like Mode0 or Mode1 must be added in Canvas->Title->Buttons. 2 buttons are added by default and it is constraint to be able to hold 1 or 2 buttons for this demo project.
    - ConfigHolder: This script is attached to the ConfigHolder GameObject and we can configure the min and max times that a game can last and also the random seed of the session. -1 to be random each time or any other value to have a repeatable seed.
    - SoundSystem: This script is attached to the SoundSystem GameObject and holds the references to the music and sfx of the game.
- Game:
    - TimerSystem: This script is attached to the SoundSystem GameObject and here we can modify from which second of the game the low-time animation starts.
    - ScoreSystem: This script is attached to the ScoreSystem GameObject and we can modify the point values of each type of match.
    - GridSystem: This script is attached to the GridSystem GameObject and we can modify:
        - Size of the fruits inside the background tile, relative to this same background tile ([0, 1], being 1 as big as the tile).
        - Width and height of the grid. It is configured to be 6x8, but can be changed. Just be careful because if a big number is selected, it may not fit on the screen on some resolutions.
- Other configs:
    - OrbesData: This is a ScriptableObject that holds which sprites correspond to each type of Orb.

4. Other Notes

- The UI elements are packed into a SpriteAtlas to make more efficient draw calls
- The Canvas used to move orbs is an individual canvas inside the root canvas, this reduces the amount of dirty canvases and therefore, the amount of updates and re-draws needed.
- In general, if a field from a class or struct can be accessed and modified by other objects, and the getter/setter or Property would be a simple return or assign, I made them public.
- A bigger amount of orbs in height is stored on the board, this is done to achieve a smooth dropdown of the orbs. So if the height of our board is 8, 8 + fixedAmount is going to be the true height of the board, but this is hidden outside of the GridCore API and can be ignored when using the API.
- In general, the code doesn't have many comments as I tried to make the functions names and contents very explicit, but if any function is bigger than usual, some directions are commented. GridCore is commented on each function to describe it, as it is the core of the game.
- The game was tested on 2 physical devices, and it worked perfectly and the UI adapted as expected. I also tested on several resolutions and aspect ratios in the Editor and also worked as expected.

5. Resources

Images found here, here and here.
Sound effects found here and here.