Integrated Gyroscope as main interaction system in a videogame for Mobile Devices

Vicent Santamarta Martinez

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

July 16, 2021

Supervised by: Jose Vicente Martí Avilés

Acknowledgments

First of all, I would like to thank my supervisor, José Vct. Martí Avilés for his eternal patience and support even when I have been a difficult student to work with. I would also like to thank my friends and family for supporting me this months and playing the game everytime I asked them to.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

Flip Blocks is a puzzle game for mobile devices where different shaped pieces will fall from the upper side of the game board and will slowly move towards the bottom. The player will have to move and rotate the pieces in order to make them fit in the board and complete lines. Pieces are built of 4 blocks each with different shapes. The pieces can stack up until they reach the top of the board, in which case it means Game Over for the player.

By completing one or more lines, the player's score will be increased, and the completed lines will disappear, making all the blocks above the erased lines fall one line. The goal of the player is to keep playing for as long as possible by deleting lines and get the highest score.

Contents

CHAPTER 1

INTRODUCTION

Contents

For this project, we will be developing a videogame for mobile devices from scratch, using the Unity3D engine. Our goal was to build a game where the device's integrated gyroscope was the core for the player interactions in the game. For that, we took Tetris[22] as our base and changed the way the player interacts with the game, adding some new interesting game mechanics to an already well known system.

1.1 Work Motivation

Gyroscopes in games are a bit tricky to work with as Input systems, and until now they have mostly been used in games where you can control a ball a move it by tilting your smartphone device. What I wanted to accomplish here as some kind of game design challenge, was to take an iconic game, which everyone knows how to play already and change it so it can only be played using the gyroscope integrated in most of the modern mobile devices.

For that task we took Tetris, which its falling and moving mechanics are already known by most people and might be a perfect fit to test the gyroscope as core Input System. Whilst the rotations might be tricky because they are usually made using a single button which can lead to interesting design approaches to this mechanic. My goal here is to develop an interesting and intuitive game, which still keeps the spirit of the original work. I hope to develop this through heavy interactions and play-testing, focusing on the overall Player experience more than in the technical bits of the game. And to not be afraid to make drastic changes to the original concept if it is not not working or is not as intuitive as intended.

1.2 Objectives

My core objective is to build a game that is intuitive and interesting, and that in the end still keeps the spirit of the original game, but still feels original. For that, my goals are:

- To develop a polished playable version of the game built upon continuous play test.
- To design a videogame where the gyroscope is the only means of interaction within the core loop system of the game.
- To make the game accessible to as many people as possible (for example incorporating accessibility options for people with color blindness)

1.3 Environment and Initial State

The first steps towards building the game were to find a concept that could fit the goals for the project. For that I started researching all the Tetris game mechanics and how they are developed. And because I wanted the game to feel as similar to the original as possible there were a few steps towards building the game, before getting hands in with the code.

The first was to find a concept that could fit within our goals. For that I started to research all about the Tetris game mechanics and how they are developed, and because I wanted the game to feel as close to the original as possible I went through all the Tetris Guidelines[4], which are a set of rules written by the developers of what a game needs to be named Tetris, and written down any concept that came to mind through that research.

The second step was to research about gyroscopes and how to work with them in Unity3D[9], I have never used them before so I needed to gather knowledge in order to build a good Player experience.

The next step was to learn about player accessibility and how to develop for people with disabilities, so I went to Game Developers Conference vault and saw any video related to building accessible games.[2]

Once I had a concept in mind that I liked and knew a bit more about gyroscopes, was time to create a blank project in Unity3D and start prototyping, while I wrote down the specific design decisions for the game.

CHAPTER 2

Planning and resources evaluation

Contents

This chapter shows the original planning for the development of the project and the resources needed for it.

2.1 Planning

In this section it is explained how the time work has been divided between the different tasks and sub-tasks necessary to get the project done. Because the project is being developed through different iterations and will go through different versions, For the sake of clarity, it has been decided to make a table were you can see the total time invested in each of the iteration versions, and the tasks that are being done through them (see Table 2.1). Also, how tasks have been planned and distributed through time can be found in Figure 2.1.

Task	Estimated duration (hours)
Prototype version	46 hours
Prepare Unity project, repository and GDD	6 hours
Core gameplay development	30 hours
Basic art and audio assets	6 hours
Playtest prototype	4 hours
Alpha version	80 hours

Iteration on prototype playtest results	20 hours
Refine core gameplay implementations	40 hours
Basic menus implementation	10 hours
Implement win-lose conditions	2 hours
Decide the art and audio direction for the game	4 hours
Playtest Alpha version	4 hours
Beta version	104 hours
Iterate on Alpha playtest results	20 hours
Refine art assets	16 hours
Refine audio assets	16 hours
Implement visual effects and player feedback	24 hours
Implement main, setting and max score menu	12 hours
Implement data saving	12 hours
Playtest Beta version	4 hours
Release version	20 hours
Iterate on Beta playtest results	20 hours
Thesis tasks	50 hours
Thesis writing	40 hours
Thesis presentation	10 hours
Total	**300 hours**

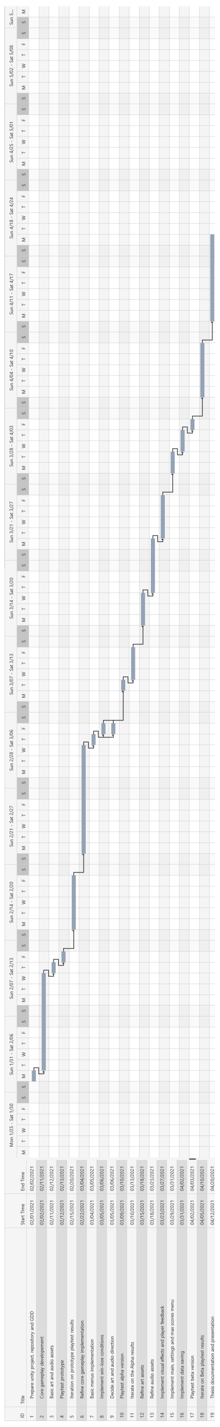
Table 2.1: Task time distribution

2.2 Resource Evaluation

In this section is going to focus on analyzing and listing every human and technological resource that we are going to need for the development of this project.

2.2.1 Human Resources

For estimating the costs of the human resources it is going to be assumed that the game is being developed by only one developer, building himself the technological and visual parts of the game and we will be assuming that the developer is based on Spain. Knowing this information and through a quick search in indeed[8], can be seen that the average salary for a junior software developer in Spain is 19,266€ per year which makes it around 9.64€ per hour. Knowing that the project is only 300 hours long, the total cost of the developer to build the game would be of 2,892€ for the development of the whole project.



ID	Title	Start Time	End Time
1	Prepare unity project, repository and GDD	02/01/2021	02/02/2021
2	Core gameplay developement	02/02/2021	02/11/2021
3	Basic art and audio assets	02/11/2021	02/12/2021
4	Playtest prototype	02/12/2021	02/13/2021
5	Iteration on prototype playtest results	02/15/2021	02/20/2021
6	Refine core gameplay implementation	02/22/2021	03/04/2021
7	Basic menus implementation	03/04/2021	03/05/2021
8	Implement win-lose conditions	03/05/2021	03/06/2021
9	Decide art and audio direction	03/05/2021	03/06/2021
10	Playtest alpha version	03/09/2021	03/10/2021
11	Iterate on the Alpha results	03/10/2021	03/13/2021
12	Refine art assets	03/15/2021	03/18/2021
13	Refine audio assets	03/18/2021	03/23/2021
14	Implement visual effects and player feedback	03/23/2021	03/27/2021
15	Implement main, settings and max scores menu	03/29/2021	03/31/2021
16	Implement data saving	03/31/2021	04/02/2021
17	Playtest beta version	04/02/2021	04/03/2021
18	Iterate on Beta playtest results	04/05/2021	04/10/2021
19	Thesis documentation and presentation	04/12/2021	04/20/2021

Figure 2.1: Example of a Gantt chart (made with online visual paradigm)

2.2.2 Technological Resources

As for the technological resources, it is going to be analyzed the software and hardware resources needed for the development of the game.

- **Hardware:** For the hardware it will only be needed two specific things, first a computer that can run Unity3D to develop the game, and secondly, a mobile device that has an incorporated gyroscope so the game can be test, it does not need to be really fancy:
  - Smartphone Xiaomi mi A2 Lite: **189€**.
  - Laptop Asus Rog G513: **698€**.
    - Processor: AMD Ryzen 7 4800H.
    - Graphic Card: Nvidia GTX 1650
    - Memory: 15GB.
    - Internal Memory: 512GB SSD.
- **Software:** It is going to be listed every software that we are using to develop the game:
  - **Unity3D 2020.3.8.f1 (Free):** This is the engine in which the game is going to be built.[19]
  - **Visual Studio 2019 Community (Free):** This software comes together with unity and its the IDE that used to write the code in Unity3d.[10]
  - **GitHub Destop (Free):** GitHub is going to be the repository for the game project.[6]
  - **Visual Paradigm (Free):** Online UML tool used to create the diagrams for this document.[13]
  - **Overleaf (Free):** Online LaTex tool used to write this document.[12]
  - **Adobe Illustrator 2019 (19,66 €/month):** Software used to develop the 2D graphics for the game.[1]
  - **DoTween (Free):** Unity package library that helps to build basic animations through code.[3]
  - **Unity Custom Hierarchy (Free):** Unity package that lets you customize the Editor hierarchy to make game development easier.[5]
  - **Notion (free):** Tool used to build the Game Design Document and organize work [11]

Item	Cost
Human costs	2.892,00 €
Tech resources	926,32 €
Hardware	887,00 €
Laptop	698,00 €
Mobile device	189,00 €
Software	39,32 €
Adobe Illustrator	39,32 €
Total	**3.818,32 €**

Table 2.2: Total costs for the project

With all the resources needed for the project listed, now an estimation of the projects can be done (See table2.2). All of the costs are are really straight forward to calculate with the data from the resource analysis, but for the software costs. Because the project is slightly longer than one month working 8 hours a day, it is required to get two moths of license, that is why the price for Adobe Illustrator is twice the one that was presented in the analysis.

Chapter 3

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design. In this chapter we are going to learn the requirement analysis, design and architecture for the project. We will be also learning about the interface design and some of the decisions behind them.

3.1 Requirement Analysis

In order to carry out the requirements analysis for the project it is important to identify the challenges have to be overcome for the development of the game and accomplish the goals set for the project. So let's focus on knowing a bit more about the game so the challenges that have to be overcome.

Flip Blocks is a *Puzzle* video game where the player is given a set of different shaped pieces that fall from the top of the playfield towards the bottom and have to place them in the playfield. The playfield is formed by a board with and array with a fixed number of columns and lines. Pieces fall one(1) line every second, and when it arrives to the bottom, or falls on top of another piece, the piece gets place and the player is given a new random piece. If one piece is placed above the top of the playfield then it is game over for the player.

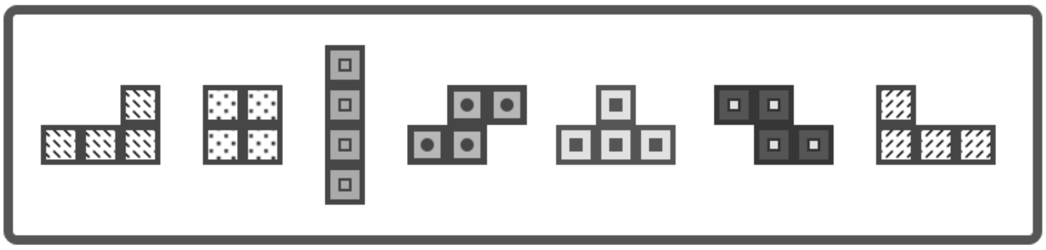


Figure 3.1: Pieces used for Flip Blocks

On the game start the Main Menu is loaded. In the main menu the player can find the options *Play*, *turn audio On/Off*, *turn vibration On/Off* and *(Exit Game)*. Pressing on the later will close the game. Selecting the vibration or the sound buttons will change the state of the selected button, turning on and off the audio or vibration. Pressing on the *Play* button will let the player start playing the game.

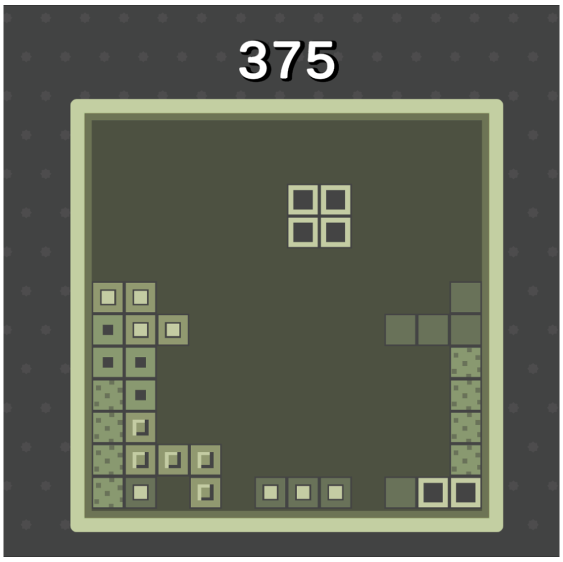


Figure 3.2: How the playfield looks on the project

Once the player has started the game will receive a random piece from the set (See Figure 3.1) and it will instantly start to fall one line every second. While the piece is falling, the player can move it tilting the device right or left, or rotate it by rotating the mobile device. This actions are detected using the mobile device incorporated gyroscope. When the piece falls on the last row of the playfield, or on top of another piece then the piece is placed (See figure 3.2. If the placed piece goes beyond the top of the playfield then the player looses the game and its game over. Otherwise, the player receives a new random piece. In the flowchart in figure 3.3 can be seen how the system handles this operations.

While playing, the player can pause the game at any time, freezing the game and opening the pause menu. On the pause menu the player can *turn audio On/Off*, *turn vibration On/Off*, *Resume Game* and *Exit*. The later option finishes the current game and transports the player to the main menu. The audio and vibration buttons work the same way as in the main menu. And finally the resume game option hides the pause menu and lets the player to keep playing the same match that were playing.

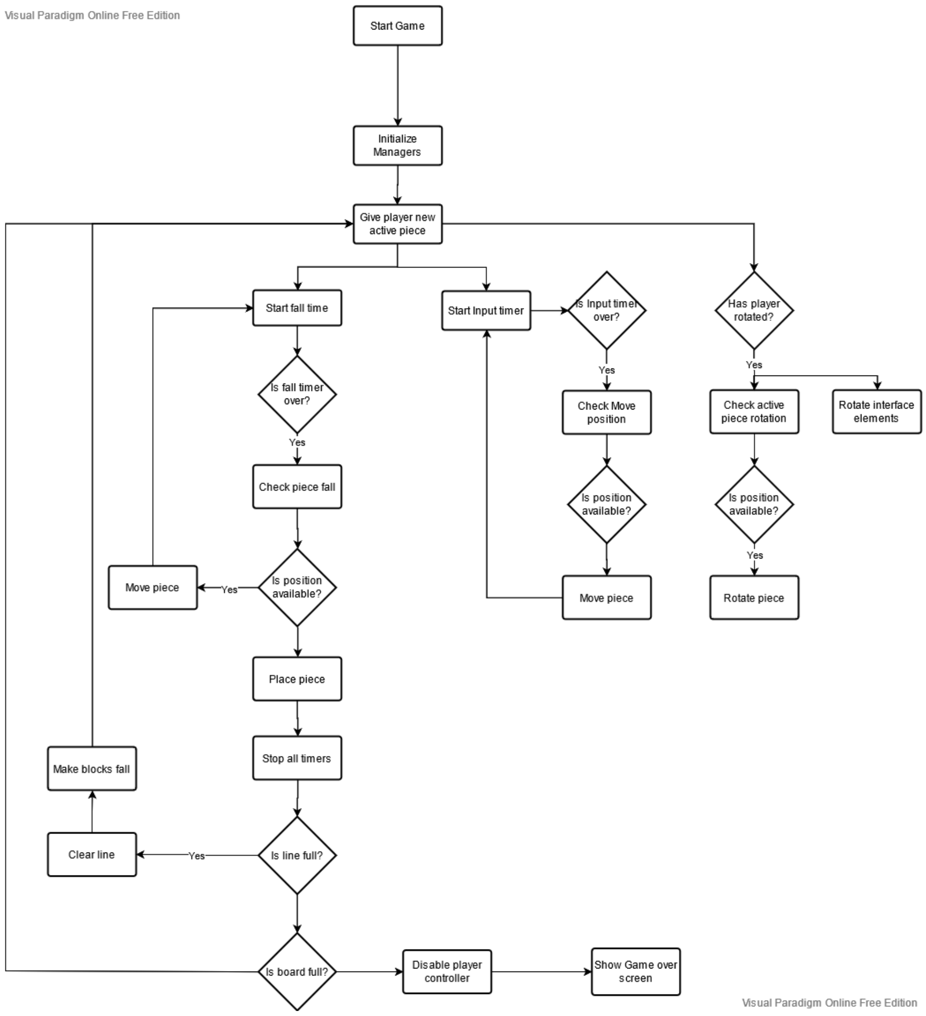


Figure 3.3: Core gameplay flowchart.

For all the player actions, the game will send back feedback through visual effects, sound cues, and vibration patterns. And because player is constantly rotating and moving the the phone, the game's User Interface will adapt so the game can be played from any orientation.

3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. This function is described as a set of inputs, its behavior, and its outputs. For our exact project they would be:

- **R1.** The player can mute or unmute the sound effects and game music.
- **R2.** The player can turn on and off the vibration feedback.
- **R3.** The player can start the game by pressing "Play".
- **R4.** The player can close the game by pressing "Exit".
- **R5.** The player can move the piece by tilting the phone towards the sides.
- **R6.** The player can rotate the piece by rotating their mobile device in the direction they want the piece to rotate.
- **R7.** The System will be able to adapt the games User Interface to the player's device orientation.
- **R8.** The player can clear lines by filling them with the pieces.
- **R9.** The player can pause the game at any given time by pressing the pause button found on the lower right corner of the screen.
- **R10.** The player can exit to the main menu by pressing the "Exit" button found in the Pause Menu.
- **R11.** The player can resume their game by pressing the "Resume" button found in the Pause Menu.
- **R12.** The player can access the settings menu.
- **R13.** The player can see their final score when the game has finished.
- **R14.** The player can play again by pressing the "Play Again" button found in the Game Over screen.
- **R15.** The player can return to the main menu by pressing the "Exit" button found on the Game Over menu.

3.1.2 Non-functional Requirements

- **R16.** The User Interface must be understandable no matter in which orientation the player is holding the mobile device.
- **R17.** The controls have to be intuitive and easy to learn.
- **R18.** The visuals of the game must be accessible for players with visual disabilities.
- **R19.** When the player has performed and action the player has to be able to identify what has happened.
- **R20.** The game state should not change while interacting with menus.

3.2 System Design

In this section will be presented the system design that is going to be developed for the project. In the following pages are defined all the cases of uses that solve the requirements found in the functional requirements, and a case of used diagram.

As for the case use diagram, it can be seen that most of the actions the player can perform are quite straight forward, but for the move piece and rotate piece actions ha a dependency of the system. That is because when the player moves or rotates, the system has to check if the new positions that the blocks are going to occupy are available, and if they are not, the player will not be able to make that move or that rotation. (See figure 3.4)

3.3 System Architecture

The requirements needed to play the game are really low and almost any modern smartphone run it.

- **Hardware:** The mobile device needs to have and integrated gyroscope. It can be found in almost any modern smartphone, but there might be some cases which does not include it.
- **Software:** The mobile device must run at least the Android 4.4 "Kit Kat" version or higher. It is the minimum requirement for a phone to run a Unity game.

3.4 Interface Design

Because for Flip Blocks the player has to be continuously rotating the mobile device, so they can rotate the pieces, the game needs to adapt to those changes so it can be played in a comfortable way. The first element of the game that has been affected by this feature is the game Playfield. The playfield is squared instead of rectangular as in

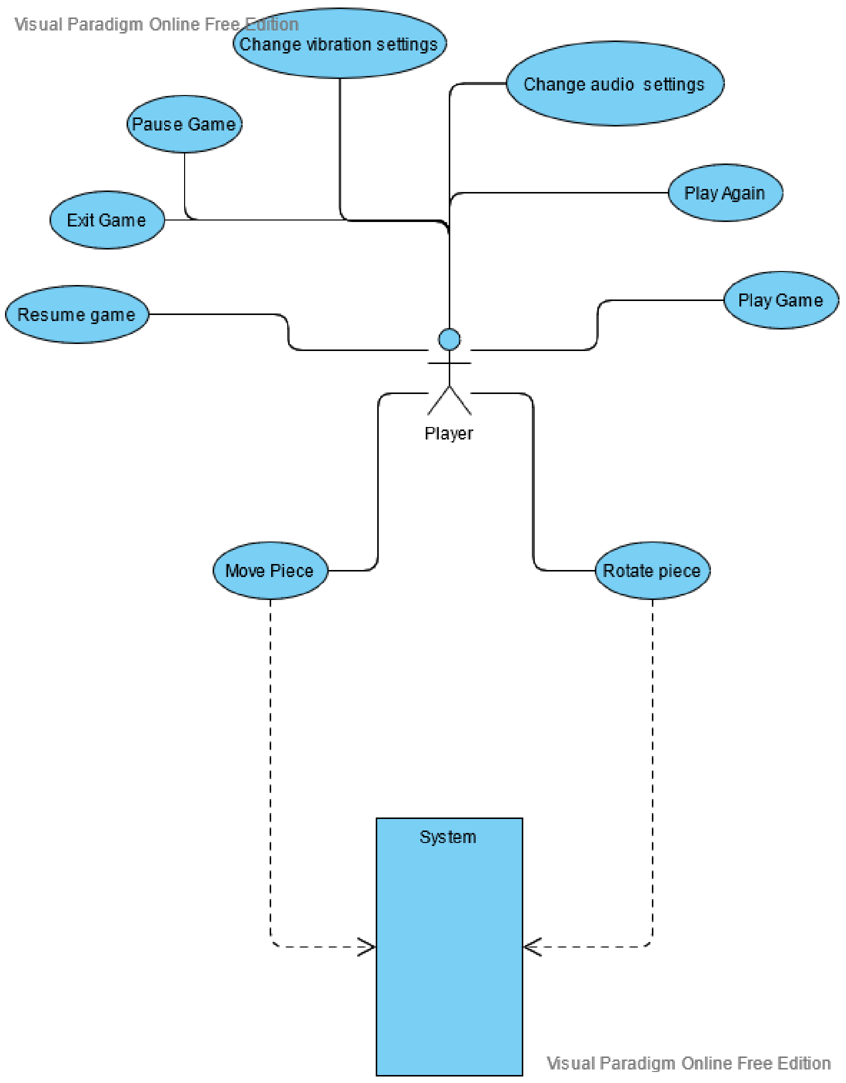


Figure 3.4: Use Case diagram (made with Visual Paradigm Online)

the traditional Tetris. That is because the playfield rotates with the player's device, so the player is always seeing the piece falling from the top to the bottom, a rectangular playfield would have to change its size when changing from a horizontal to a vertical orientation, whilst the squared always keeps its size no matter the orientation.

The second element is the user interface, which needs to adapt to those changes in the orientation of the screen. This does not happen on menus because the text direction forces the player to keep their phone in a certain position to read comfortably. So this section of the report will focus on the user interface for the core loop, when the player is placing pieces.

Adaptive User Interface

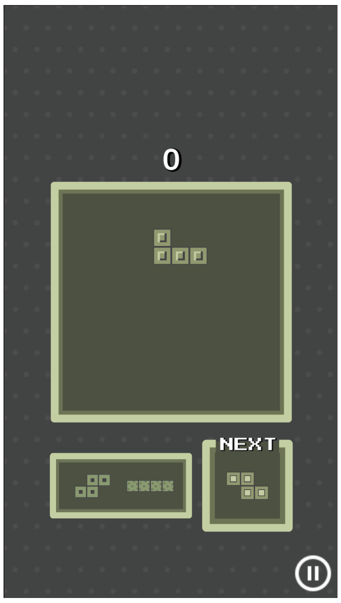


Figure 3.5: User Interface in protrait.

The Figure 3.5 shows how the game looks with the J piece in the board when the player is holding their phone phone straight, what is known as the portrait orientation. If the player would want to rotate the piece towards the left, leaving the shorter part of the shape of the piece at the bottom and the larger part at the right, the player have to rotate their phone in that direction. Once the phone has been rotated the player would see the game as in the Figure 4.4. This is not really comfortable to play with, so now all the interface elements need to adapt to the new orientation.

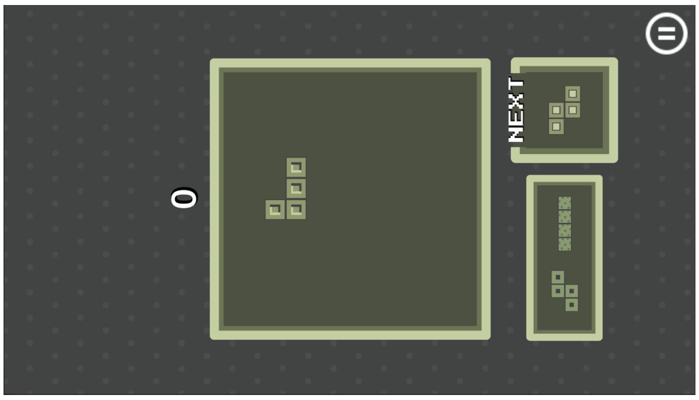


Figure 3.6: User Interface in protrait, rotated.

The Figure 3.7 is a screenshot of how the user interface looks when it adapts to the devices orientation. For these adaptive interface three (3) types of interface elements have been developed so they can adapt to each of the device's orientation in its own way. The first type, are the ones that follow the rotation of the screen. This way no matter how the player is holding the device, they will always look like they are straight up. An example we can see on the figures is the preview of the next piece which will always look up no matter the rotation.

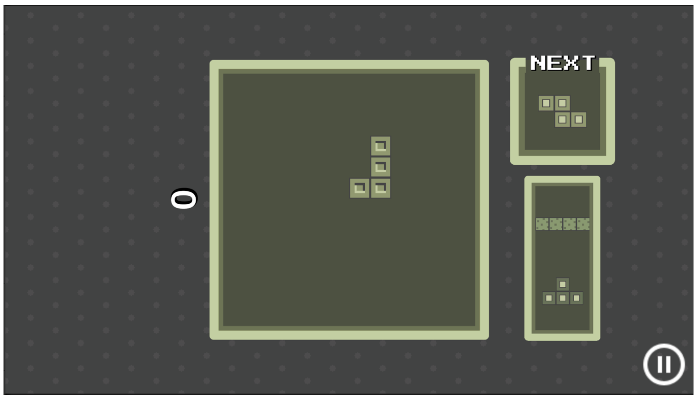


Figure 3.7: User Interface in landscape.

The second type of adaptive that we can find in the game are the ones that have an independent rotation for each of the orientations. In our game, that is the score text, which we can see in the image that it has not rotated. This is because for composition reasons we don't want them to rotate, but there are some orientations we would like them to rotate so they can always be comfortably read. (See Figure 3.8).

The third and last type of adaptive interface element we can find here are the ones that move with the screen rotation. This is used in elements that we want to keep in a certain position of the screen no matter how the player is holding the device. In the previous figures, we can see the pause button, which no matter what rotation the game is having, the player can always find it in the lower-right corner of the screen.

Next pieces preview

One of the trickiest features in term on User Interface was to find how to show the player the next pieces, and that it could be understood no matter which orientation the player is holding the device. The figure 3.9 the game is using a next piece preview similar to the rest of the Tetris games. In this preview, the player would understand that the next piece is the J piece on the left, and the last piece would be the L piece on the right. But if the player now rotates the device towards the left, now the J piece on the left would be placed on the bottom and the L piece on the top, which on a first glance for the player, would mean that the next piece is now the L piece on the top and the last piece the J on the bottom.

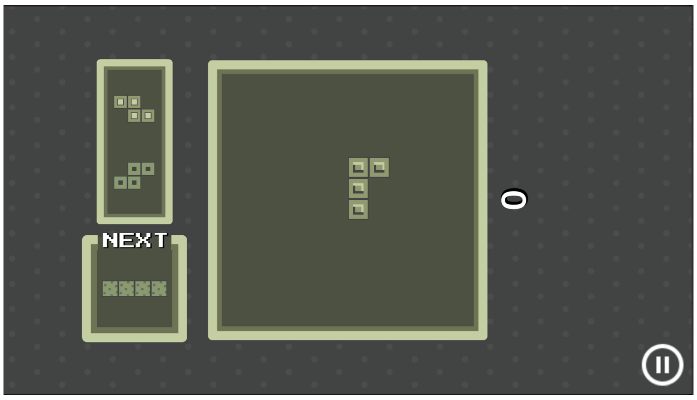


Figure 3.8: User Interface in inverse landscape.

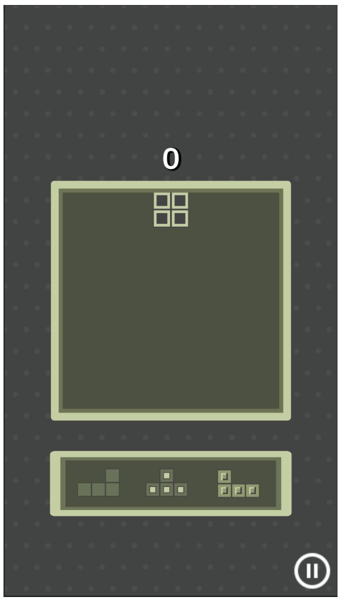


Figure 3.9: User Interface without visual Hierarchy.

A solution for this could be a little move animation with the rotation of the device but it would feel really weird seeing the pieces changing positions and shuffling around on the preview while playing. Instead it was decided to build a visual hierarchy around the directly next. In the Figures 3.7, 3.5 and 3.8 show how the next piece is separated from the other two and its slightly bigger centering the player attention on the directly next piece. This is so the player can instantly know which is going to be the next piece and from the distance relation it will be easier to know which one will come after, this way the player always has a reference no matter the rotation and will always know how the next pieces are going to come.

3.5 Designing for Accessibility

When we want to making the game accessible for as many people there are a countless number that you can incorporate to the game, in this section it is going to be explained how the game has been designed in order to make it more accessible to everyone.

3.5.1 Designing for visual disabilities

When designing for visual disabilities it is always good to keep in mind that there are different types of visual disabilities.

- **Designing for color blindness:** When designing to color blindness there are different kinds of color blindness that affects how people interact with colors. For that reason it is always recommended when designing the visuals for a game to focus on contrast and patterns to differentiate the different elements existing in the game. For that reason, the game has been developed using a single color palette and have focused heavily on patterns to differentiate the pieces between themselves.

- **Designing for shortsighted disabilities:** People with short sight problems are really common so it was important to make the game comfortable for them to play too. To accomplish that, the original board size, which was of 14 by 14, so less pieces had to fit on the screen. Also, the elements on screen have been made as big as possible so they can be easier to be seen.

3.5.2 Designing for auditive disabilities

In order to make the game more accessible for people with auditive impairments it was important to make every action the player does to give heavy visual or haptic feedback. So whenever the player moves the pieces, places the piece, clears a line or collides against one of the corners of the playfield there will be an animation, visual effect or vibration pattern that will tell the player what has happened.

3.5.3 Finding the right resources

When trying to incorporate accessibility in your game it might be hard sometimes to find what is the right way to approach it. Luckily accessibility has been a really important matter through the past years for game developers, and those developers have shared their research and knowledge on the internet. Here can be found some places to start when trying to incorporate accessibility as part of your game:

- **GDC Vault[2]:** Is a well renown website where many developers share their respective knowledge on their areas of expertise, including accessibility om games.

- **Game Maker's Toolkit[14]:** A YouTube channel sharing all kinds of knowledge about game design. And it is also developing a series on designing for disability.

- **Game accessibility guidelines[7]:** A collaborative effort between a group of studios, specialists and academics, to produce a straightforward developer friendly reference for ways to avoid unnecessarily excluding players, and ensure that games are just as fun for as wide a range of people as possible.

Requirements:	R4
Actor:	Player
Description:	At the start of the game the player is presented with the option to press "Start Game" to start the game from the Main Menu
Preconditions:	1. The player is on the Main Menu
Normal sequence:	1. The player selects "Start Game". 2. The game loads the CoreLoop scene 3. The player is given a first active piece. 4. Control is given to the player.
Alternative sequence:	None.

Table 3.1: Functional requirement «CU01. Play game»

Requirements:	R1
Actor:	Player
Description:	The player mutes or unmutes the game audio
Preconditions:	1. The player is on the Main Menu or in the Pause Menu
Normal sequence:	1. The player presses on the speaker icon. 2. If the audio es on, then the audio manager turns it off. 3. Change the icon to audio muted.
Alternative sequence:	2.1. If the audio is off, the audio manager turns on the audio. 2.2. Change the icon to audio on.

Table 3.2: Functional requirement «CU02. Change audio settings»

Requirements:	R5
Actor:	Player
Description:	The player moves the active piece left and right to let it fall in the desired position
Preconditions:	1. The player has to be in the CoreLoop scene. 2. The game must not be paused. 3. The game must not be over. 4. The player has to have an active piece.
Normal sequence:	1. The player tilts the mobile device to the right or left. 2. The system checks if the new position in the movement direction is available. 3. If the position is available, piece moves to the new position.
Alternative sequence:	2.1. If the position is not available the piece stays in the same position

Table 3.3: Functional requirement «CU03. Move Piece»

Requirements:	R6
Actor:	Player
Description:	The player rotates the active piece so it fits in the desired position of the board
Preconditions:	1. The player is on the CoreLoop scene. 2. the game must not be paused. 3. The game must be not over. 4. The player must have an active piece.
Normal sequence:	1. The player rotates the phone in one direction. 2. The system checks if the rotation is possible. 3. If the piece is not colliding with anything then the piece rotates
Alternative sequence:	3.1. If the positions are not available the system checks if there is a nearby available position. 3.2. If there is a nearby position, the piece rotates and moves to the new position. 3.1.1. If there is no available position, the piece cannot rotate and stays the same.

Table 3.4: Functional requirement «CU04. Rotate Piece»

Requirements:	R9
Actor:	Player
Description:	The player pauses the current game
Preconditions:	1. The player has to be on Core Loop scene. 2. The game must not be paused.
Normal sequence:	1. The player presses the pause button on the lower-right corner of the screen. 2. The game freezes its current state, not updating the fall and input timers. 3. The gyroscope control is disabled for the player. 4. The Pause menu is displayed on screen.
Alternative sequence:	None.

Table 3.5: Functional requirement «CU05. Pause Game»

Requirements:	R10
Actor:	Player
Description:	The player cancels the current game and goes back to the main scene
Preconditions:	1. The player has to be on the Core Loop scene. 2. The game has to be paused and the main menu on screen.
Normal sequence:	1. The player presses on the exit button. 2. The Game manager loads the Main Menu scene.
Alternative sequence:	None.

Table 3.6: Functional requirement «CU06. Exit to Main Menu»

Requirements:	R10
Actor:	Player
Description:	The player resumes the paused game
Preconditions:	1. The player has to be on the Core Loop scene. 2. The game has to be paused and the main menu on screen. 3. The device must have the same orientation as when it was paused.
Normal sequence:	1. The player presses resume button. 2. The pause button hides. 3. Input and Fall timers are resumed. 4. The player is given its gyroscope control back
Alternative sequence:	None.

Table 3.7: Functional requirement «CU07. Resume Game»

Requirements:	R2
Actor:	Player
Description:	The player turns on/off the vibration feedback
Preconditions:	1. The player is on the Main Menu or the Pause Menu
Normal sequence:	1. The player presses the vibration icon. 2. If the vibration is on, then the Vibrator turns it off. 3. Changes the icon to vibration off.
Alternative sequence:	2.1. If the vibration is turned off the it gets turned on. 2.2. Changes the icon to vibration on.

Table 3.8: Functional requirement «CU08. Activate/Deactivate vibration»

Requirements:	R14
Actor:	Player
Description:	The player play the game all over again
Preconditions:	1. The game has to be over. 2. The player has to be in the Game Over screen.
Normal sequence:	1. The player press the button "Play Again". 2. The Game Over screen hides. 3. The Game Manager loads the Core Loop scene
Alternative sequence:	None.

Table 3.9: Functional requirement «CU09. Replay Game»

Chapter 4

Work Development and Results

Contents

This section is a dissection of how the project has been developed from the start to the end. It will expose how the project has changed through iteration, playtesting and how close is the final result to the project goals.

4.1 Work Development

The work development will be exposed in chronological order so it can be correctly appreciated how playtest and iteration affected the development of the game for the good and for the bad.

4.1.1 The original concept (prototype version)

At the beginning of the project, when the Game Design Document was built, the game the concept was really different. Though it was still based on Tetris, the player could only move the pieces to the left or to the right. In order for the player to rotate the piece, it had to collide them against the blocks of already placed pieces or against a special block placed on the playfield called obstacles (See Figure 4.1.

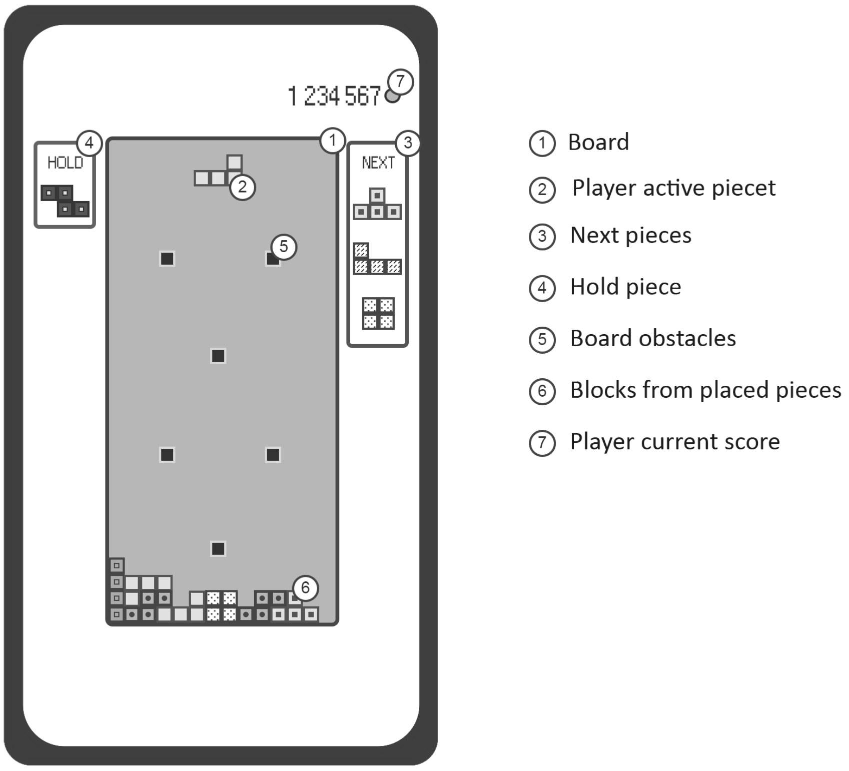
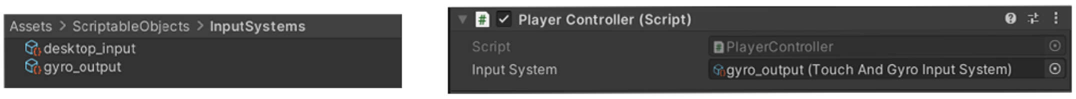


Figure 4.1: Original concept schematic

Moving the pieces

The first tasks where to implement the core gameplay mechanics, starting by making the piece move to left and right in the screen. For that, a *PlayerController* class was built, which will control the player's active piece, make it fall, move it, and rotate it according to the player input. One of the goals for the project is to focus the development on iteration and playtesting (See section 1.2), so it was needed a set of modular and reusable input systems that the *PlayerController* could simply read the input from without having to worry what is happening behind the curtains.

Those *InputSystems* where built using the Unity3D ScriptableObjects [17]. ScriptableObjects are a special type of unity objects that are really useful to store data and to build functionabilities that do not require of being components and the best part is that they can be easily interchangeable in the Unity Editor.


Assets > ScriptableObjects > InputSystems
desktop_input
gyro_output

Player Controller (Script)
Script PlayerController
Input System gyro_output (Touch And Gyro Input System)

Figure 4.2: InputSystem objects in the Unity Editor

To build the structure for the different input systems, a parent abstract class *InputSystem* was built. This inherits from *ScriptableObject* so the any of the implemented *InputSystems* only have to extend the abstract class InputSystem. This was developed this way because it was important that all the Input System that had to be developed needed to be scriptable Objects, and by extending the abstract class it is forcing them to be scriptable objects too. The best part of this is structure, is that the input systems can be easily changed just by drag and dropping it in the component's reference. In the figure 4.2 can be seen how the InputSystem scriptable objects look in the Unity Editor. In this figure, the *PlayerController* is reading the input from the gyro input system. If a change to the desktop input system was needed, the developer would only have to drag and drop the game object into the PlayerController reference.

Placing the pieces

With the Input System structure implemented, now it was time to make the pieces fall from the top of the playfield towards the bottom, placing the piece when it arrives to the bottom. The implementation for this was really straightforward, it was only needed to tell the PlayerController to instantiate a piece in the spawn position and make it fall one line every certain time. When the piece arrives to the edge of the board, tell the PlayfieldManager to place the piece, spawning Blocks objects in the piece position, and hiding the piece to be used later.

With the pieces placement, now the system needs to, every time a piece moves, check if the position is available or not. For that a class that will check, in the direction in which we are trying to move the piece, if the new position for the piece is available was built, it was called MoveAttepmt. For the player controller, it only needs to create an MoveAttempt object, pass the active piece and the move direction. The MoveAttempt class will manage internally the move, checking whether the piece can move or not. The

PlayerController has direct access to the move result data, this way the PlayerController script only needs to focus on reading the player input and applying it to the active piece. (See Listing 4.1)

```
    private bool MoveActivePiece(Vector2 moveDir)
    {
        MoveAttempt moveAttempt = new MoveAttempt(moveDir, _activePiece);
        _activePiece.Move(moveAttempt.position);

        if (Mathf.Abs(moveDir.x) > 0) //Vibrate if horixontal move
            Vibrator.CreateOneShot(50, 50);

        return moveAttempt.moveState == MOVE_STATE.SUCCESS;
    }
```

Listing 4.1: Move function from the player controller

In this time we also made some changes to the *PlayerController*. Up until now the *PlayerController* read got the data from the input system every frame, this is okay when you want a smooth movement, but we wanted to make the movement a lot more like in the original Tetris, where the player can only move the pieces every fraction of a second. To do that I used the same Timer class that made the piece fall. It is a really simple class that calls a delegate[15] after the time given to it has passed.

Rotating the pieces

Now that the pieces could be moved and placed, it was time for the tricky part of the mechanics, rotating pieces. In the original concept the player had to make the active piece collide against already placed pieces or against a special set blocks called obstacles that were static in the playfield (See Figure 4.3). As it has been seen through the project, each class have been focusing on their own work trying to be as independent as possible. This way classes are easier to read, understand and modify.

So like as with the *MoveAttempt*, a *RotationAttempt* class was built in order to handle piece rotation checks. This new class only received the move direction, the active piece, and the points of collision. The direction and collisions points with placed blocks were important for the collision checks because the rotation direction depended on where the collision happened along the piece shape and the movement direction.

Implementing the gyroscope input

Once we had the piece moving, rotating and being placed, it was time to test the game in the mobile device to see how it felt to be played with a gyroscope. For that we needed

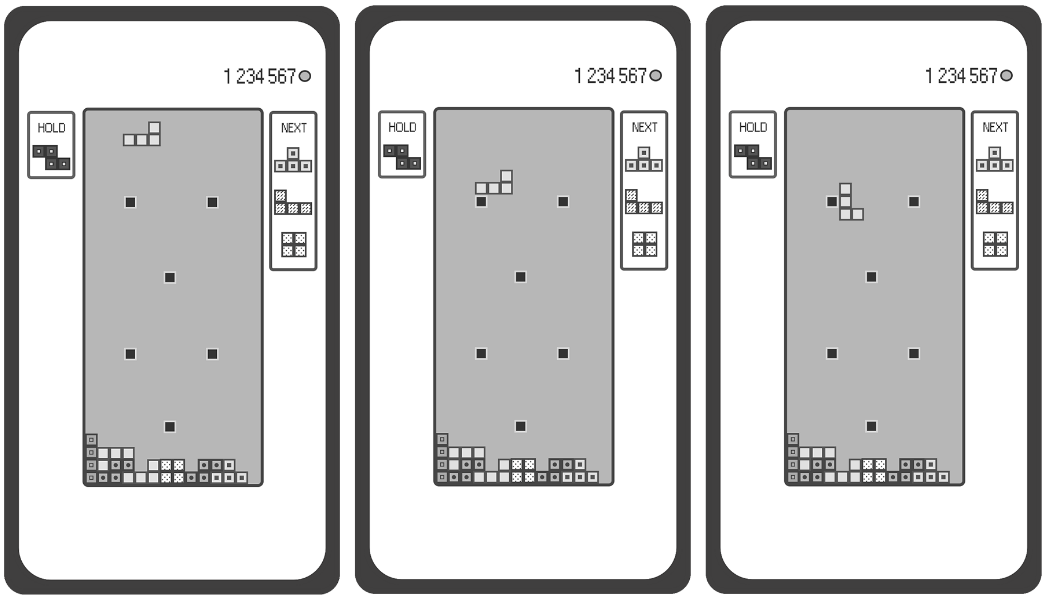


Figure 4.3: Mock up of the rotation in the original concept

to access the Unity Gyroscope[20] class to get the device's rotation data. The *Gyroscope* class has three(3) sets of data that can be useful for our purposes:

1. **attitude:** Returns the attitude (overall rotations) of the device.
2. **rotationRate:** Returns how much the phone has rotated since the last interval.
3. **userAcceleration:** Returns the acceleration that the user is giving to the device.

Because the *userAcceleration* only gives the acceleration and finding the actual rotation of the device would be too hard and inefficient, this option was discarded this one really early on the implementation, between the *rotationRate* and the *attitude* was slightly harder to chose. With the *rotationRate* it was needed to keep track of the player rotation all the time, a class that had to keep on updating the rotation was need, and so a *RotationManager* was built. On the other hand working with the attitude was quite easy because it told the current rotation of the device, so the system always knew in which direction the phone was tilted towards. After a bit of testing of both ti was decided to use the attitude for its facilities towards knowing the rotation state of the mobile device. With this decided the input system only needed to translate the data so the *PlayerController* could understand it and it was almost ready for testing.

Giving the player pieces

With the input ready to work, it was time to give the player random pieces that spawn on the top of the playfield and fall towards the bottom. For that I created two classes, the

PieceBase which would store the piece data (block sprite and the piece menu preview) and would do any operation needed by the piece specially related to how the visuals of the piece game object (ex. adapting to the size of the board, or moving/rotating the actual game object). The *PieceBase* also stores a discrete version of the piece in the shape of a matrix. And a *PieceManager* which will be the one that will give the player random pieces.

For the random generation of the pieces I we used the so-called "7-bag random generator" that can be found in the Tetris Guidelines. The player is given two bags with all 7 pieces randomly shuffled inside and the player starts by extracting pieces of one of them. When one of them gets empty, the player is given a new bag of pieces.

In Flip Blocks, this is hadled by the *PieceManage*, which internally has two(2) queues, the main one, from which the player extracts the pieces from, and the other queue that serves as the backup to the main one. At the start of the game both queues are filled and shuffled. When the player extracts a piece, the main queue extracts one piece and gives it to the player whilst the backup queue extracts one of its pieces and puts it in the main queue. When the backup queue gets emptied, it is refilled with shuffled pieces again. Randomizing the pieces like this, help to make that the player can rarely get two consecutive repeated pieces, and if they do, it only happens once every two bags in the worst case.

Playtesting the original concept

With the core mechanics working, it was time to start playtesting the game using the gyroscope incorporated in the mobile device. Using Unity Remote technology[18] helped in testing the game often because it removed the need to build the game every time a slight adjustment to the Input sensibility or game speed was made.

After playtesting the game and making small adjustments trying to find a sweet spot where the game felt nice to play, A few problems were found with game's concept, making the game unable to accomplish the goals for the project:

- Rotating the piece was not intuitive and in fact it was really difficult to complete lines.
- Obstacles in the center of the screen made it so if the player failed to rotate the piece and placed it on top of the obstacle instead, it created an obstruction and it was almost impossible to clear lines below the obstructed piece.
- Opening the obstacles only to the sides made it to difficult to rotate pieces in the direction that the player wanted, and the player was moving the piece left and right all the time.
- Making the playfield bigger so there was more space and it was harder to obstruct the board made it really boring to play, and there was too much space to fill.

Because in the end there were too many downsides to this concept, and the kind of interactions that it was creating were not the ones that I wanted to accomplish, I decided to scrap this concept and work with a new one that could fit better my goals and objectives.

4.1.2 Iterating on the game concept

With the original concept scrapped the project needed to find a new system that could accomplish its goals, so it was time to take pen and paper and start to ideate. A little later a new concept was found that was worth to start implementing right a way. Still like as in the previous concept the game would receive a piece that would start to fall towards the bottom and the player has to move and rotate to piece in order to fill lines and clear them. But this time the player had to rotate the piece by rotating the mobile device in the direction that they want it to turn.

Thanks to that the system developed was really modular, most of the systems built for the first concept prototype were still useful for the new concept. Only needed to scrap the rotation system and build a new one using the same system that was being used before in the *InputSystem* to know how the device was being tilted. The input system was already using the horizontal and vertical axis of the gyroscope data, so no it only needed to add the 'z' axis to know how the device was being rotated.

Developing the new rotation

Some early problems arised from using the *attitude* for knowing the current device's rotation, and it is that the z-axis value of the gyroscope is really dependant to the values found in the x-axis and y-axis. The difference in the z-axis of having the device rotated 90 degrees in the y-axis or -90 degrees in the y-axis was of more than 60 degrees, which cause big inconsistencies playing the game. To solve that it was needed to use the gyroscope's *rotationRate* for the input in the z axis. The old *RotationManager* was now reused with a few changes so every class could access it to know the current rotation of the device and it incorporated an event that would tell any listener when the device has been rotated. With the new rotation working, it was time to playtest the game again and see how it played out.

After playing with new system for a while found that it felt intuitive and fun to play with, but because the player is constantly rotating the mobile device, the game requires some changes to how it reacted to the player input. Otherwise the game would feel weird and uncomfortable to play (See Figure 4.4).

Adapting to the player rotation

The main characteristic of a Tetris game is that the piece always starts at the top of the board and falls towards the bottom. To keep that characteristic even when the player is rotating their mobile devicem, the playfield needed to rotate with the player. Rotating the actual playfield would take too much work because when moving the piece the system

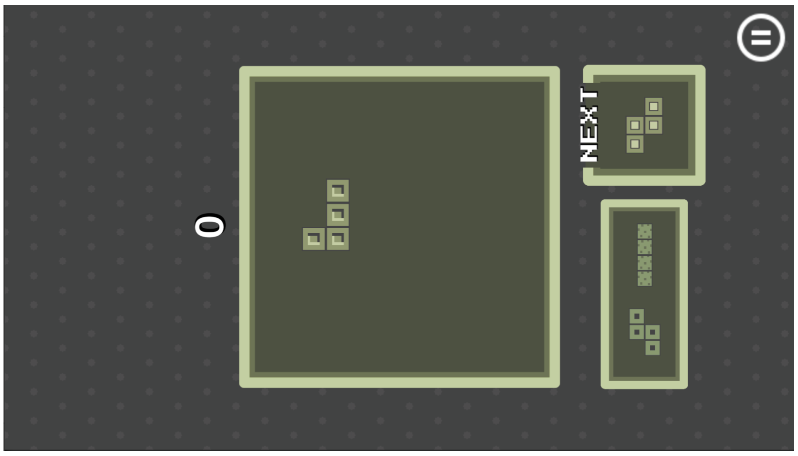


Figure 4.4: User Interface in protrait, rotated.

needs to have in mind the current rotation of the player. So a trick was used instead, the game made the camera that was showing the playfield to use a render texture[16]. A render texture is a special kind of texture that allows the developer to show in real time what is happening in front of a camera that is not rendering directly to the screen. The render texture, was then applied to a Raw Image object that was placed in the user interface, so what the player is looking at is just an image of the playfield, and not the playfield itself.

Now the user interface only have to make this image rotate with the screen, and even when playfield itself is the same, the player is seeing the pieces start at the top and move to the bottom no matter the rotation. To make the user interface elements rotate automatically with the screen rotation a small component was built, which it automatically subscribes to the rotation event from the *RotationManager* and rotates the piece. That means that any piece that I want to rotate, I only have to add this component and will instantly rotate with the screens orientation, see section 3.4 to see how the user interface work.

Another consequence that the rotating had on the playfield is that keeping it rectangular made no sense because it had to change its size to adapt to the device's orientation. So it was necessary to make the board squared instead of rectangular so it would keep its size no matter what the player rotation was (See Figure 4.5).

With this approach the *PlayerController* now needed to change what axis it is reading depending on the devices orientation, when the device was in Portrait on Inverse-portrait the horizontal axis is the x-axis, while in Landscape and Inverse-landscape the horizontal is the y-axis. So a few modification were needed in order to get the right axis (See Listing 4.2).

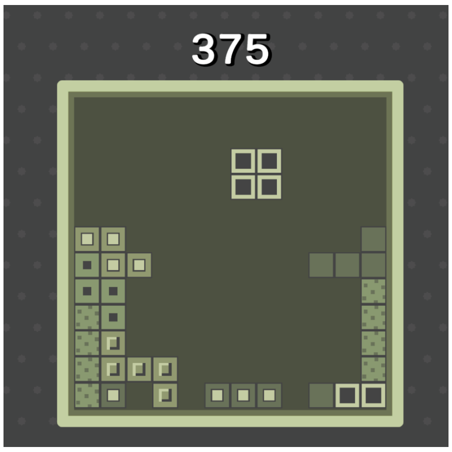


Figure 4.5: Final look of the playfield.

```
    private void OnPlayerInputTimer()
    {
        float x = 0f;
        if (_orientation == SCREEN_ORIENTATION.PORTRAIT)
            x = _inputSystem.GetHorizontal();
        else if (_orientation == SCREEN_ORIENTATION.LANDSCAPE)
            x = -_inputSystem.GetVertical();
        else if (_orientation == SCREEN_ORIENTATION.INV_PORTRAIT)
            x = -_inputSystem.GetHorizontal();
        else if (_orientation == SCREEN_ORIENTATION.INV_LANDSCAPE)
            x = _inputSystem.GetVertical();

        if (MoveActivePiece(new Vector2(x, 0f)))
            _inputReadTimer.ResetTimer(); //Restart timer if it can keep moving
        else
            OnPiecePlaced();
    }
```

Listing 4.2: How the PlayerController reads the input.

4.1.3 Beta version

After some playtesting with new concept the game felt close what it was intended to build for the project, so it was time to start developing the rest of the mechanics. This feature was not developed before because the goal was to make the prototypes to focus on how it felt to move, rotate and place the pieces. But now it was time to start making the player to clear lines and to be able to loose.

Win-lose conditions

The reference to the blocks placed in the playfield was stored in the *PlayfieldManager*, so the only thing ti was needed to do was to, when placing a piece, check the lines occupied and if they are filled, tell the blocks to be destroyed, the destroy animation played by each block independently. Once all the blocks of the lines are destroyed, an instruction is sent to make all the blocks from the lines above the ones destroyed to fall as many lines as there have been cleared.

Blocks are not really destroyed for efficiency purposes. Instead they are deactivated and will be re-used later on when a new piece is placed. This management of the objects is known as pooling[21], it is usually used when the system needs to have objects that have to be continuously instantiated and destroy from the scene. Both of this actions have a big overload in CPU times, so instead of continuously do that, the system only needs to instantiate the game objects at the beginning of the game and then just enable and disable them when it is needed.

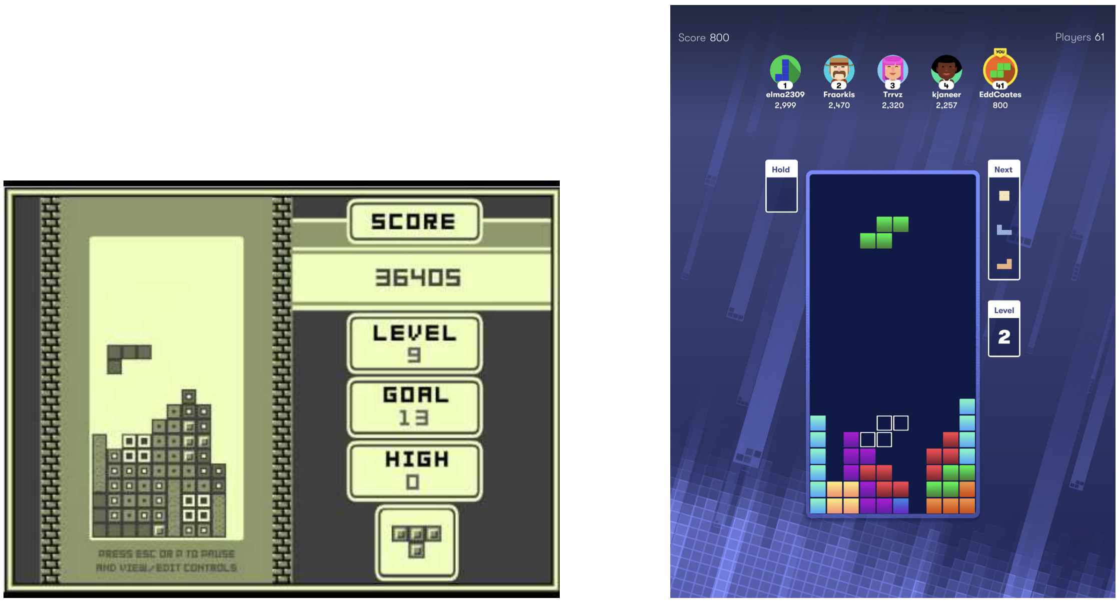
Now that the player can clear lines, the loose condition was now implemented. For this game is really easy to find out though, when the player places a piece if any of the blocks is higher than the top of the board, then it is game over and the player looses. When that happens, a small animation destroying all the blocks is triggered and after, it the game over screen is displayed to the player. In this screen the player can see the final score and can either replay the game, which tells the game manager to reload the current scene, or the player presses exit and goes to the main menu.

Building the visuals

When developing the visuals for the game, the game tried to keep a similar look to the original Tetris, but slightly more modern. For that, Some reference were taken from the modern version of Tetris (mobile) and tried to mix it with the retro looks of the Game Boy Tetris (See Figure 4.6).

While developing the visuals for the game it was important to always keep in mind the designing for disabilities guidelines (see section 3.5). One of the most important accessibility guideline is to focus the visuals on contrast and patterns instead of colors, so so the game was going to use the colors found the interface of the GameBoy Tetris and adapt it to the needs of the project which led to what can be seen in figure 4.7.

It was also important that the game's interface was easy to read, so the game needed the interface to easily adapt to the screen rotation, but thanks to the component that was previously built for rotating the playfield, making any element rotate, move or even scale depending on the device's rotation was quite easy. More about the interface design can be found in the Section 3.4.



(a) Tetris (Gameboy) (b) Tetris (Mobile)

Figure 4.6: Visual references

4.2 Results

Having in mind the objectives presented in the section 1.2 they have not only been accomplished but the shaped the approach given to the project. First because one of the core objective was to build a game through playtesting and iteration, it forced to build modular and reusable code that could always be reused to build any kind of prototype related to the game. Through playtesting it was found how the initial concept was not the kind of experience that the project wanted to accomplish, and by scraping it and iterating on it, a concept that approached way better to the project's objective was built.

As for these second objective it was have accomplished too, the final version of the game really manages to keep the Tetris spirit in a game that can only be played with the gyroscope which was really important for the project.

When designing the game for accessibility, it has affected greatly on how the game was conceived. The playfield has less rows to make the pieces bigger and easier to be seen, the user interface elements are really big so they can be seen without any problem, any action of the player has a feedback either through visual effects or vibration and the visuals have been developed focusing on using only one color palette, focus on patterns and high contrast to differentiate each element in screen. So through the project everything developed had the thinking behind about how what was being built was going to be affected by the people playing it, which was my goal. It would have been nice to have

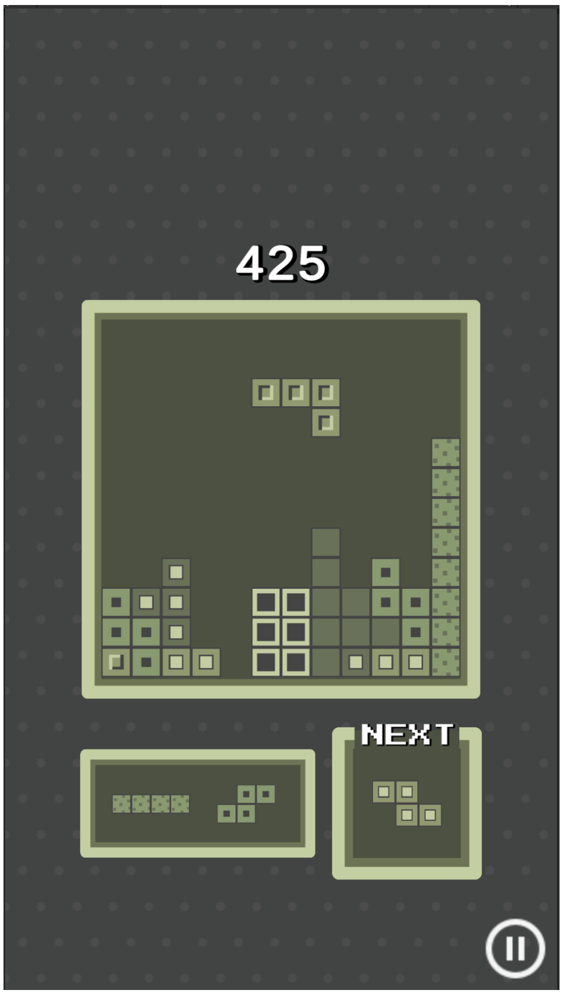


Figure 4.7: Flip block final interface design.

added an options menu with a few more accessibility options like scalable text, but because of the time constrains it has not been possible.

Finally the project source code can be accessed through this link to the repository: `https://github.com/vicentamen/tfg_gyroscope_tetris`

And the Android .apk with the demo of the game in this link: `https://drive.google.com/drive/folders/15W4IojBWIOfjrzeo9QiipWbixrYTWKEY?usp=sharing`

Chapter 5

Conclusions and Future Work

Contents

5.1 Conclusions

In conclusion building a game is hard and finding the right mechanics for your game is really important. Sometimes when building projects for university we try to make gigantic games similar to the ones we play, but always forget that those mechanics might not work and because they are so hard to build and we have invested a lot of time in them, we end up sticking to games that feel like they are not finished. With this project I wanted to a bit against that current, I wanted to find a small challenge and through iteration build the best concept that could overcome that challenge. For me that meant that the game needed building a small game because I knew that it might need to get scrapped. Even knowing that I ended up investing way too much time in the collision and rotation system of the first concept. But even after that I managed to find a concept fit better the goal of the project and through iteration i brought it to a point in which I am really satisfied with.

5.2 Future work

The game needs a little bit more of work to be fully polished. I would like to explore more visual styles for the game even if it differentiates it from Tetris. While developing this thesis report I found out about a new Unity3D input system which might be smoother than the one developed by me and I would like to experiment with it. And finally I would like to polish more the overall experience of the game to a level that the game could be published in the Android Play Store.

Bibliography

[1] Adobe. Adobe illustrator. https://www.adobe.com/es/products/illustrator.html.

[2] Game Developers Conference. Gdc vault - designing for disabilities. https://www.youtube.com/c/Gdconf/search?query=designing

[3] Demigiants. Dotween. http://dotween.demigiant.com/.

[4] Tetris fandom. Tetris guidelines. https://tetris.fandom.com/wiki/Tetris_Guideline.

[5] Febucci. Unity custom hierarchy. https://www.febucci.com/2020/10/custom-hierarchy-for-unity/.

[6] GitHub. Github. https://github.com/s.

[7] Game Accessibility Guidelines. Game accessibility guidelines. http://gameaccessibilityguidelines.com/basic/.

[8] Indeed. Junior programmer average salary in spain. https://es.indeed.com/career/programador-junior/salaries.

[9] Jjules. Unity3d gyroscope. https://forum.unity.com/threads/unity-and-the-accelerometer-vs-the-gyroscope-a-complete-guide.451496/.

[10] Micorosft. Visual studio 2019 community. https://visualstudio.microsoft.com/es/vs/community/.

[11] Notion. Notion. https://www.notion.so/.

[12] Overleaf. Overleaf. https://es.overleaf.com/project.

[13] Visual Paradigm. Visual paradigm. https://www.visual-paradigm.com/.

[14] Game Makers Toolkit. Designing for disabilities. https://www.youtube.com/watch?v=xrqdU4cZaLw.

[15] Unity3D. Delegates. https://learn.unity.com/tutorial/delegates#5c894658edbc2a0d28f48aee.

[16] Unity3D. Render texture. https://docs.unity3d.com/es/2018.4/Manual/class-RenderTexture.html.

[17] Unity3D. Scriptable objects. https://docs.unity3d.com/Manual/class-ScriptableObject.html.

[18] Unity3D. Unity remote. https://docs.unity3d.com/es/2018.4/Manual/UnityRemote5.html.

[19] Unity3D. Unity3d. https://unity.com/.

[20] Unity3D. Unity3d gyroscope manual. https://docs.unity3d.com/ScriptReference/Gyroscope.html.

[21] Wikipedia. Object pooling. https://en.wikipedia.org/wiki/Object_pool_pattern.

[22] Wikipedia. Tetris. https://es.wikipedia.org/wiki/Tetris.

Appendix A

Source Code

Because there are a lot of small components interacting in the game, instead of presenting here some lines of the code, I will add here the link to the github repository of the project so you can explore all the code with no constrains: https://github.com/vicentamen/tfg_gyroscope_tetris

And you can also find the Android .apk build of the project here https://drive.google.com/drive/folders/15W4IojBWIOfjrzeo9QiipWbixrYTWKEY?usp=sharing