Facultad de Informática

Trabajo de Fin de Grado

PASAR (Plataforma Arduino para Speedruns Automatizados y Resultados)

Por

Eva Sánchez Muñoz - Nota: 10

Héctor Calvo Gómez - Nota: 10

UNIVERSIDAD COMPLUTENSE
MADRID

Dirigido por

José Luis Vázquez Poletti

David Pacios

MADRID, 2024–2025

Abstract

This investigation presents the development of a TASbot programmed with scripts from Python that have been integrated into an Arduino's board, from where the execution is carried out. The objective of the project is to carry out the conection between a computer and an Arduino to run a program capable of performing an automated speedrun. The speedrun would be from a 2D videogame, with random components, that would allow showing the adaptability of the TASBot. This work takes advantage of several Python's graphic libraries, to be able to take screenshots, look for elements or interact with the screen of the videogame. This document would explain in detail the research done, the most important methods implemented in the development phases and the conclusions obtained.

Esta investigación presenta el desarrollo de un *TASBot* programado con *scripts* de *Python* que ha sido integrado en una placa de *Arduino*, desde donde se lanza su ejecución. El objetivo del proyecto es realizar la conexión entre un ordenador y un *Arduino* para ejecutar un programa capaz de realizar un *speedrun* automatizado. El *speedrun* será de un videojuego en 2D, con componentes de aleatoriedad, que permitan demostrar la capacidad de adaptación del *TASBot*. Este trabajo se aprovecha de varias bibliotecas gráficas de *Python*, para poder hacer capturas de pantalla, búsqueda de elementos o interacciones con la ventana del juego. El documento explicará con detenimiento las investigaciones realizadas, los métodos más importantes implementados en las fases del desarrollo y las conclusiones obtenidas.

Palabras Clave: Arduino, OpenCV, TASBot, RNG, Speedrun, Speedrunner, Python, pyWin32, PyAutoGUI, Subprocess, PyTesseract, Balatro.

Índice general

Dedicatoria

Para empezar quiero agradecer a mis tutores, David Pacios 'Pascal' y José Luis Vázquez Poletti por dejarme unirme al proyecto y habernos guiado tan bien durante este viaje a mi compañera Eva, por aceptar que me una en una idea de TFG tan divertida, por aguantar mis momentos de agobio, y por acompañarme en la crianza de y el tratamiento de discalculia 'pepito.py'.
A mis mejores amigas Espe, Ainhoa y Rocío por ser la voz de la tranquilidad y la razón en momentos donde tenía ganas de abandonar, por esas llamadas viendo musicales o jugando 'Overwatch' o yendo a conciertos y lo más importante, por animarme a seguir adelante
A ASCII y LAG por acompañarme , por tantas risas y momentos que guardare en mi corazón y por hacer de la facultad a veces mi segunda casa, gracias sobre todo a Alec, Cris, Luis, Victor, Zero, Canelada, Ali, Pablo, Sai, Luque y Jaime
Y sobre todo gracias a mis padres y a mi hermana , por creer en mí, y seguir apoyándome durante estos años tan complicados. Muchas gracias a todos.

'Ohana means family family means nobody gets left behind or forgotten'
-Héctor

En primer lugar, quiero dar las gracias a mis tutores, David Pacios 'Pascal' y José Luis Vázquez Poletti, por haberme permitido realizar este proyecto. A mi compañero, Héctor, por acompañarme tanto en los buenos como en los malos momentos del desarrollo, en especial durante la crianza y el tratamiento de discalculia de 'pepito.py'. A mis padres y a mi hermana, por apoyar mi decisión de cursar esta carrera y seguir haciéndolo después de tantos años. A mis amigos, Diego y Álvaro, por seguir proponiendo planes para desconectar después de tantos años. Y a todas las personas que he conocido durante mi travesía por esta universidad, en especial a la gente de ASCII, por dejarme ser yo misma sin miedo.

Porque dudo que hubiera llegado tan lejos sin vosotros, vuestros mensajes de ánimo y vuestro apoyo. Muchísimas gracias por todo.

"For those who come after"

-Eva Sánchez Muñoz

Capítulo 1. Introduction

1.1. History of speedruns

The concept of speedrun (shown in Figure 2.2) was created as a modality for various developers of videogames that wanted their players to end the playthroughs [1] in a certain amount of time, for example *the Drag Race* (Kee Games, 1977), *Metroid* (Nintendo, 1986) or *Prince of Persia* (Broderbund, 1989). Even with the cases mentioned, this mode didn't capture the attention of the community until the release of *Doom* (id Software, 1993).

In 1993, the videogame *Doom* (id Software, 1993) was published with timers that marked the time transcurred since the beginning of the level and a repeat function that works with files called lumps to be able to visualize the whole match later, allowing the player to publish their replays on the Internet [2].

A year later, in 1994, Frank Stajano created the *Doom Honorific Titles*, a catalog of titles that could be gained after completing a series of challenges in the game, and Simon Widlake published COMPET-N, a web page that worked as a leaderboard for the players.

In 1997, Nolan Pflug created NSD (Nightmare Speed Demos), a web page that keeps track of the different speedruns finishing by publishing the same year a whole game replay of the game *Quake* (id Software, 1996), in a game made by numerous speedruns called *Quake Done Quick* and in 1998, Nolan published a web page with the name Speed Demos Archive that allowed to save demos [3] of the done speedruns.

These speedruns gave place to some techniques that are still used in the present, for example the bunny hopping [1] a method used to gain velocity through the mechanic of jumping.

In the year 2003, it began to circulate throughout the Internet a video called *Super Mario Bros. 3 Time Attack Video* shown in Figure 2.1. The video in question had a duration of eleven minutes, time that the player called 'Morimoto' used to complete *Super Mario Bros. 3* (Nintendo, 1988) without

[1] Complete Game of a videogame from start to finish
[2] Global and decentralized computer network
[3] Small free sections in videogames sent by the developers as a demonstration of their titles

dying to obtain in a more easy way 99 lives.

Later that year it was discovered that to do that video, a NES or *Nintendo Entertainment System* (Nintendo, 1985) emulator called 'Famtasia' was used. That emulator executed a file that contained the sequence of pulsations of buttons and save points that allowed resetting different states of games, to obtain the best results, obtaining with it the perfect playthrough .

The file used received the name of TAS (shown in Section 4.1) but was frowned upon by the community who considered it 'cheating'. As a result, the 'Morimoto' speedrun was declared fraudulent.

That same year, a web page was created called TASVideos with the intention of calming down the community of speedrunners [4] giving different warnings about the TAS files used. With those warnings, it was intended that those who viewed the speedruns that were shown on the screen understand that it was not possible to do by a human.

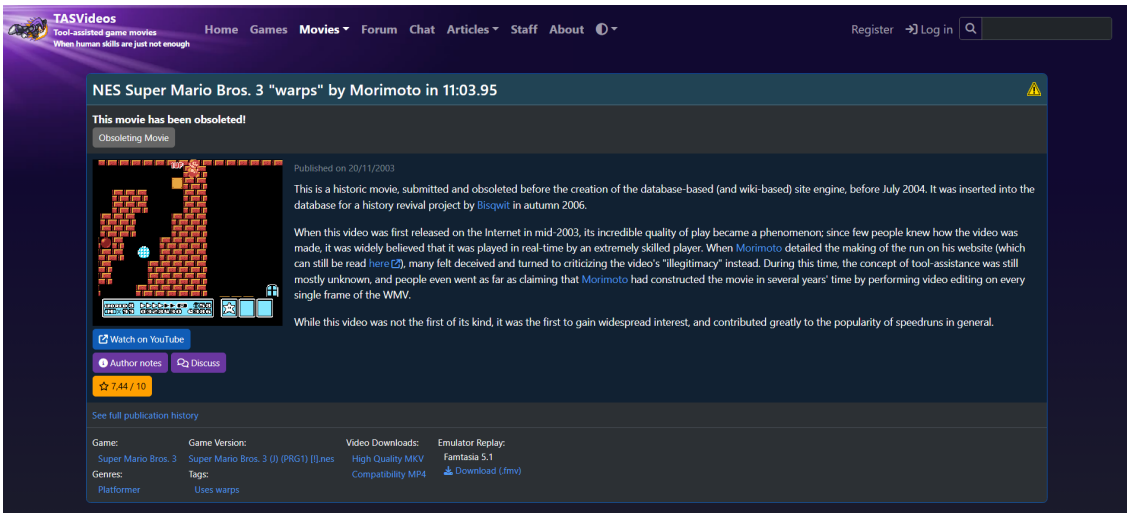

Figura 1.1: Dedicated Setion that was made with TASBot of *Super Mario Bros. 3* of 'Morimoto' in the web page called TASVideos.org

As time went on, the number of speedrunners increased and with that, the quantity of tools that could be used for the speedruns. The computers that were used had more and more resources, which allowed the emulators to work with more efficiency without losing the execution velocity.

These circumstances provided in 2006 the design of the TAS file that could be executed without modifying the previous behaviour of the console. The intention behind that concept of obtaining the exact

[4] People who dedicated to get over challenges in the least possible amount of time in videogames

simulation of each component of a console included a memory map based on 'game cartridges' and chips [5] of expansion, which in theory, if the emulator is capable of using the dependencies of the system with efficiency, it should be able to gain the same results of a speedrun made by a TAS file in a console as the results obtained in a computer.

Thanks to the deterministic behaviour of most of the consoles, which only depend on the introduction of commands by the player, it's possible to use TAS files included in games that have components of RNG (Random Number Generator). This algorithm is further detailed later when we talk about RNG algorythim (shown in Section 4.2), because it's always a sequence of pulsations of buttons that correspond with the state of the game that was generated.

In 2009, it started the development of a device that was capable of sending commands of input [6], that were recorded in a file beforehand through the ports of a video game console *NES* (Nintendo, 1985).

That same year, a hacker [7] called 'Jaku' created a tool that was capable of playing the first level of *Super Mario Bros.* (Nintendo, 1985).

But it wasn't until 2011 that the completion of a whole gamethrough was achieved thanks to the emulator FCEUX. Also, a user called 'SoulCal' created a tool that could do complete playthroughs of games from the Nintendo 64 (Nintendo, 1996), especially the video game *Super Mario 64* (Nintendo, 1996).

All in all, the upswing of popularity of the speedruns finally came after a collection of benefic funds where various speedrunners participated in a 24-hour marathon called *Classic Games Done Quick*. Thanks to the benefices obtained, the organizers decided to repeat the event on several occasions, allowing different types of existing speedruns to have better acceptance, including the ones realized with TAS.

1.2. Motivation

A speedrun[2] is a modality of game where the players try to overcome a challenge established by the developers (or in some cases, the same players) in the smallest amount of time possible. As such, speedruns can be considered an efficient method of extending the average life of a video game, resulting in players trying to use every possible mechanism, the

[5] Small structures made with semiconductor material use to fabricated electronic circuits
[6] Data which is introduced inside of a system, such as a computer or a videogame console
[7] Person who forces an operative system looking for vulnerabilities

game reaching a greater audience, and the developers detecting design failures in order to resolve them.

Even with the good parts, when the modality of speedruns is the center of the conversation, it's possible that the players get to think if the practice could be considered inapropiated by the use of cheats or the change in the original intention of the game.

Given that videogames are developed using specific code, there are numerous elements that can be used to overcome the obstacles designed by the programmer. For example, in the game *Hollow Knight* (Team Cherry, 2017) there are different zones of the map that can be visited to obtain the upgrades of the character that are needed to end the game, but a player that knows everything or the ins and outs of the game and that has enough ability can overcome the challenge with only unlocking four of the thirteen abilities available.

Having said that , numerous videogames whose significance can be lost by performing a speedrun. For example, in the game *Pokemon Brilliant Diamond/Pokemon Shining Pearl* (Nintendo, 2021) the player must be able to win against all Pokémon gym trainers, upgrading their team in the process with the objective of obtaining the title of Champion of the League. In speedruns of the game, it's possible to ignore all the matches till the Champion, ending the game in less than a quarter of an hour.

Because of this, the same players determine what is and is not permitted during a speedrun, establishing the objectives that can be achieved. These restrictions allow the speedruns to group them into different categories, those being:

- ***Any %***: The speedruns that belongs to this category can use *glitches*[8] to overcome the challenges imposed by the developers,the players are able to go beyond the limits of the maps, gain infinite munition or ignore the requisities to overcome an event, one of the most representative examples in this category is the actual world record in *Super Mario 64* (Nintendo, 1996), without stars where the player uses the mechanic of jump and collision to ignore the requisites needed to get to the final boss.
- **100 %**: In this category, the player must complete the videogame with all the objectives completed or in case that the community considers

[8]Unexpected behaviours for the developers in the middle of the programming process that leads to unwanted results

it important, only those that are needed to obtain the maximum required. An example is the colaborative set of speedruns of the game *Quake* (id Software, 1996) that compose *Quake done Quick*, where the players beat the game in maximun difficulty achieving all the secrets and eliminating all the enemies in the levels.

- ***Low %***: The speedruns that goes in this group force the players to overcome an objective with the least amount of tools acquired possible. This condition includes the methods of transaction in the game, such as money, items, abilities and sometimes, player levels. For that, this type of speedruns, is common that the players take advantage of the game design errors and they use glitches. An example for this category is the actual world record in the *The Legend of Zelda: Twilight Princess* (Nintendo, 2006) where the player uses a glitch that allows to go through the collisions of the map when the player holds a rupy [9] shown in the Figure 1.2.

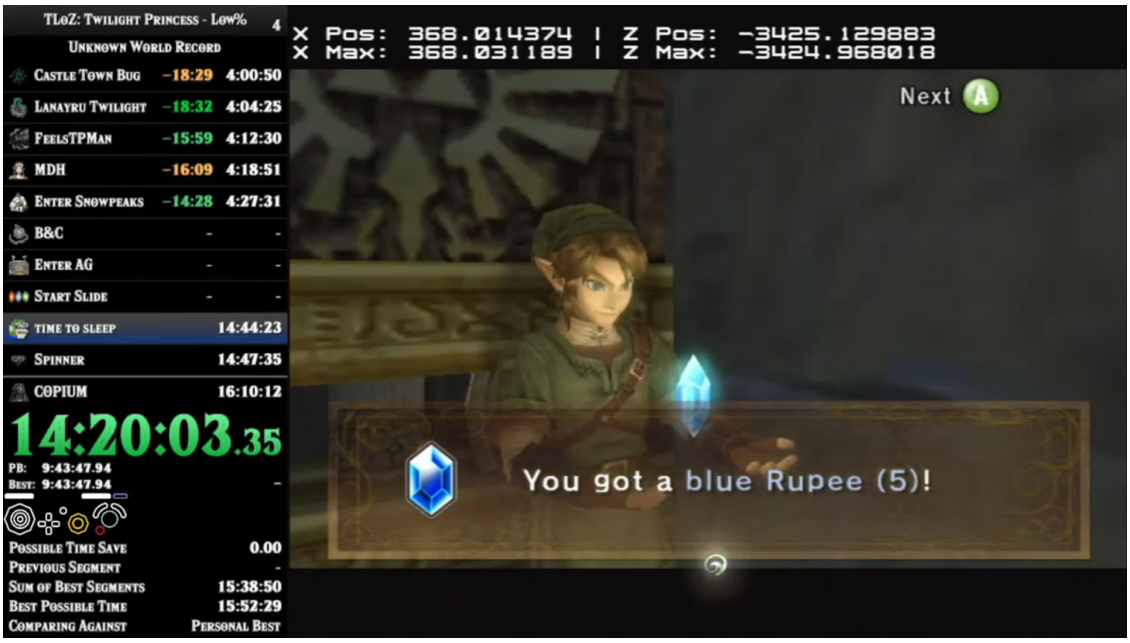

Figura 1.2: Speedrun Low % of the game The Legend of Zelda: Twilight Princess in which the player takes advantadge of an animation glitch to hold a rupy and be able to go through the limits of the map

Regardless of the imposed objectives, the speedruns can also be grouped following which flow of game they utilize. The categories are the following:

[9] Game that comes from The Legend of Zelda's videogames which works like money

- ***Time Attack (TA)***: The players must exceed objectives stablished without taking into account the time used during the load screen time and the reboot of the system.
- ***Real Time Attack (RTA)***: Unlike what happens in the speedruns category of before, the speedruns from this category takes account of the loading screen and the time during the reboot of the system, having the player to search solutions to be able to optimice this events.
- ***Spliced***: In this modality, the players can overcome the objectives in a fragmented way, trying to get the least quantity of global time. Because of that, the players can repeat various times the differents challenges, using the global calculation of the speedrun, the minimun time used in each section.
- ***Tool-Assisted Speedrun (TAS)***: They are considered perfect speedruns,as they are games run by scripts[10] or an emulator, which execute a series of commands within time periods impossible to achieve using human reactions.

This last category of speedruns, realized with *Tool-Assisted Speedrun* they are the main attention of this proposal, because it seeks to design a tool that makes it easier to do speedruns through an Arduino board [11] that will contain the necessary code.

1.3. Objectives

The objectives proposed are focused on realizing an investigation about the connectivity between an *Arduino* and a computer, with the objective to be able to perform an automated *speedrun*.

All those objectives can be grouped in two big categories: research labor and development of the project.

In relation to the research labor, the objectives that we have established are:

- Conduct research into *speedruns*, and its relation with the industry of videogames and its role on the creation of the *TASBots*.

[10] Set of instructions written in a programming language to be able to automate tasks
[11] An open source electronic platform which has easy-to-use Software and Hardware

- Make an studio about the different libraries of programming , doing emphasis in the grafic libraries developed with *Python* with the purpose of achieving the tools needed to create a *TASBot*.
- Contrast which is the best way to realize a simple conection between the *Arduino* with the computer, allowing to do an exchange of data in both ways correctly.

In relation to the part of development, the proposed objectives are:

- Program a functional code with *Python* that allows to establish a connection with the *Arduino* and realize the data exchange between computer and the *Arduino*.
- Design an algoryithm that could launch the game refered in the *TASBot*, in the case that has not been detected the process, starting in the *Arduino* and then execute it in the *TASBot*.
- Develop our own *TASBot* that could realize a *speedrun* of the highly 2D videogame with big components of randomness, having the *Arduino* as control point to be able to execute the *scripts* needed for its operation.
- Once the development is finalized, perform a series of recordings that shows the cycle of the game automated that finalizes with the *TASBot* overcoming the imposed challenge.

Capítulo 2. Introducción

2.1. Historia de los speedruns

El concepto de *speedrun* (mostrado en la Figura 2.2) surgió como una modalidad creada por varios desarrolladores de videojuegos que querían que sus jugadores finalizaran sus *playthroughs*[1] en un tiempo determinado, siendo algunos ejemplos el *Drag Race* (Kee Games, 1977), *Metroid* (Nintendo, 1986) o *Prince of Persia* (Broderbund, 1989). Pese a los ejemplos, la modalidad no logró captar el foco de atención de la comunidad hasta el lanzamiento del *Doom* (id Software, 1993).

En el año 1993, el videojuego de *Doom* (id Software, 1993) fue publicado con temporizadores, que marcaban el tiempo que había transcurrido desde que comenzó el nivel, y una función de repetición, que utilizaba archivos llamados *lumps* o *.lmp* para poder visualizar el transcurso de una partida después, permitiendo que los jugadores pudieran publicar sus partidas en *Internet* [2].

Un año después, en 1994, Frank Stajano creó el *Doom Honorific Titles*, un catálogo de títulos que se podía conseguir después de realizar retos en el juego, y Simon Widlake publicó *COMPET-N*, una página web que funcionaba como tabla de clasificación para los jugadores.

En 1997, Nolan Pflug creó *NSD (Nightmare Speed Demos)*, una página web que realizaba el seguimiento de diversos *speedruns*, terminando por publicar ese mismo año una partida completa del juego *Quake* (id Software, 1996), en un video compuesto por numerosos *speedruns* llamado *Quake Done Quick* y, en 1998, Nolan publicó una página con el nombre de *Speed Demos Archive* que permitía albergar *demos* [3] de los *speedruns* realizados.

Estos *speedruns* dieron lugar a técnicas que siguen siendo empleadas en el presente, como por ejemplo el *bunny hopping* [1], un método utilizado para ganar velocidad a través de la mecánica de saltos.

En el año 2003 empezó a circular por Internet un video llamado *Super Mario Bros. 3 Time Attack Video* como en la Figura 2.1. El video en cuestión tenía una duración de once minutos, tiempo que utilizaba el jugador,

[1] Partida completa de un videojuego desde que comienza el juego hasta que finaliza

[2] Red informática mundial y descentralizada

[3] Pequeñas secciones gratuitas en los videojuegos distribuidas por los desarrolladores como demostración de los títulos

llamado Morimoto, para completar *Super Mario Bros. 3* (Nintendo, 1988) sin morir tras obtener con facilidad 99 vidas.

Poco tiempo después, se descubrió que, para la realización del video, se había usado un emulador para *NES* o *Nintendo Entertainment System* (Nintendo, 1985) llamado *Famtasia*. El mencionado emulador ejecutó un archivo que contenía una secuencia de pulsaciones de botón y puntos de guardado que permitían reiniciar el estado del juego para obtener mejores resultados, consiguiendo con ello una partida perfecta.

El fichero empleado recibía el nombre de *TAS* (se mostrará en la Sección 4.1) y su utilización, la comunidad de videojuegos no pensaba que fuera adecuada, opinaba que cualquier método utilizado para superar los límites humanos del jugador era considerado hacer trampas y, por ello, declararon el *speedrun* de Morimoto como no válido.

Ese mismo año, se creó una página web llamada *TASVideos* con la intención de apaciguar a la comunidad de *speedrunners* [4] proporcionando diferentes advertencias acerca de los archivos *TAS* empleados. Con estos avisos, se pretendía que aquellos que vieran los *speedruns* supieran que el video que se mostraba en la pantalla no era posible hacerse con habilidades humanas.

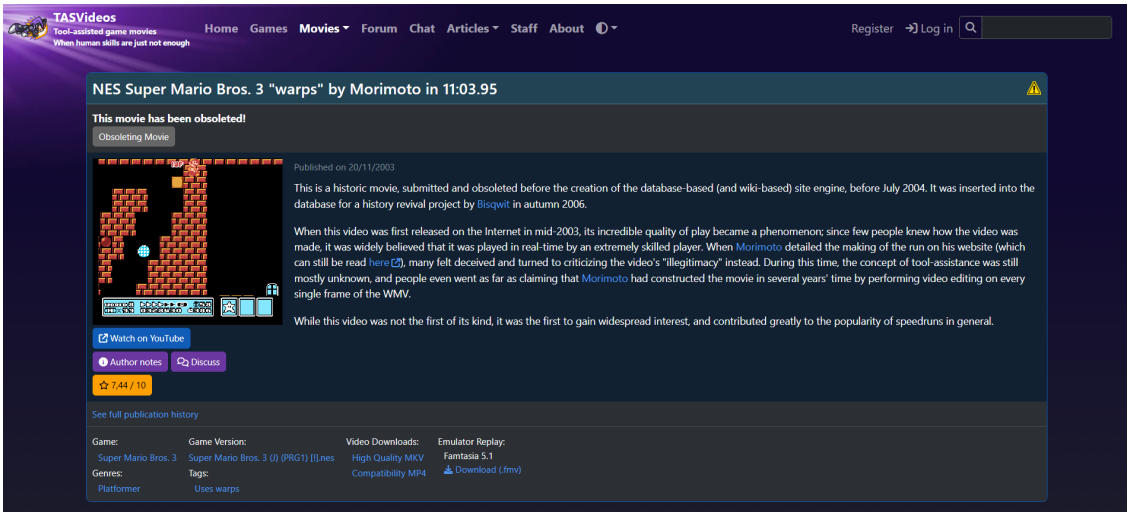

Figura 2.1: Sección dedicada al speedrun realizado con TASBot de Super Mario Bros. 3 de Morimoto en la página web llamada TASVideos

Con el transcurso del tiempo, el número de personas que realizaban estos desafíos aumentó y, con ello, la cantidad de herramientas que podían emplearse para realizar *speedruns*. Los ordenadores empleados tenían cada

[4]Personas que se dedican a superar desafíos en videojuegos en la menor cantidad de tiempo posible

vez más recursos, lo que permitía que los emuladores funcionaran con mayor eficiencia, sin perder velocidad de ejecución.

Estas circunstancias propiciaron en 2006 el diseño de un fichero *TAS* que pudiera ejecutarse sin modificar el comportamiento previo de la consola. La intención detrás del susodicho concepto era obtener una emulación exacta de cada componente de una consola que incluía un mapeo de memoria basado en cartuchos y *chips*[5] de expansión porque, en teoría, si un emulador es capaz de utilizar las dependencias del sistema con eficacia, debería ser posible conseguir los mismos resultados de un *speedrun* realizado con ayuda de un archivo *TAS* en una consola que los resultados obtenidos en un ordenador.

A raíz del comportamiento determinista de la mayoría de consolas, que solo dependen de la introducción de comandos por parte del jugador, es posible utilizar ficheros *TAS* incluso en juegos que poseen componentes *RNG (Random Number Generator)*. Este algoritmo se detalla en la Sección 4.2, porque siempre habrá una secuencia que coincida con las pulsaciones de botón hechas por parte del jugador.

En 2009, se comenzó a desarrollar un dispositivo capaz de transmitir comandos de *input* [6], que habían sido grabados en un fichero de forma previa, a través de los puertos de un controlador de una consola *NES* (Nintendo, 1985).

Ese mismo año, un *hacker* [7] llamado 'Jaku' creó una herramienta capaz de jugar el primer nivel de *Super Mario Bros.* (Nintendo, 1985), pero no fue hasta 2011 cuando logró completar una partida completa con ayuda del emulador *FCEUX*. Por otro lado, un usuario llamado 'SoulCal' creó una herramienta para poder realizar *playthroughs* completas de juegos de la *Nintendo 64* (Nintendo, 1996), destacando el videojuego *Super Mario 64* (Nintendo, 1996).

A pesar de los sucesos mencionados, el aumento en popularidad por los *speedruns* no llegó hasta después de una recaudación de fondos benéfica, donde varios *speedrunners* participaron en una maratón de 24 horas llamada *Classic Games Done Quick* [3]. Gracias a los beneficios obtenidos, los organizadores decidieron repetir en varias ocasiones el evento, permitiendo que los diferentes tipos de *speedruns* existentes en la época tuvieran mejor aceptación, incluidos aquellos realizados con *TAS*.

[5] Pequeñas estructuras de material semiconductor utilizadas para fabricar circuitos electrónicos
[6] Datos que son introducidos en un sistema como puede ser un ordenador o consola
[7] Persona que fuerza un sistema informático en busca de vulnerabilidades

2.2. Motivación

Un **speedrun** [2] es una modalidad de juego donde los jugadores intentan superar un desafío establecido por los desarrolladores (o, en algunos casos, los propios jugadores) en la menor cantidad de tiempo posible. Por ello, los *speedruns* se consideran como un buen método para extender la vida media de un videojuego, logrando que los jugadores aprovechen al máximo las mecánicas establecidas, que el juego consiga un público mayor y que los desarrolladores puedan detectar fallos de diseño para, en caso de ser necesario, solucionarlos.

A pesar de los puntos positivos, cuando se habla de la modalidad de los *speedruns*, es posible que los jugadores lleguen a plantearse si su práctica podría considerarse inapropiada por el uso de trucos o la tergiversación de la intención original del juego.

Los videojuegos, al ser un producto desarrollado con un código concreto, tienen varios elementos que pueden ser aprovechados por los jugadores para eludir los obstáculos propuestos por el programador. Por ejemplo, en el juego *Hollow Knight* (Team Cherry, 2017) existen numerosas zonas del mapa que se deben visitar para poder obtener las mejoras de personaje necesarias para terminar el videojuego, pero un jugador que conozca los entresijos del juego y tenga suficiente habilidad, es capaz de superar el desafío descubriendo solo cuatro de las trece habilidades disponibles.

A raíz de lo mencionado, existen numerosos videojuegos cuyo significado puede llegar a perderse por culpa de la realización de un *speedrun*. Por ejemplo, en el juego *Pokemon Brilliant Diamond/Pokemon Shining Pearl* (Nintendo, 2021) el jugador debe superar a todos los líderes de gimnasio *Pokemon*, fortaleciendo su propio equipo en el proceso, con el fin de poder apropiarse del título de Campeón de la Liga. En un *speedrun* del juego, por otro lado, es posible ignorar todos los enfrentamientos hasta el Campeón, terminando el juego en menos de un cuarto de hora.

Por estos motivos, son los propios jugadores los que determinan qué puede o no estar permitido durante la realización de un *speedrun*, estableciendo los objetivos que deben cumplirse. Estas restricciones permiten agrupar a los *speedruns* en diferentes categorías, siendo estos:

- ***Any %***: Los *speedruns* de esta categoría pueden hacer uso de *glitches* [8]

[8] Comportamientos inesperados por parte de los desarrolladores durante el proceso de programación que produce resultados no intencionados

para pasarse los desafíos marcados por los desarrolladores, pudiendo estos hacer que los jugadores atraviesen los límites del mapa, consigan munición infinita o ignoren los requisitos para superar un evento. Uno de los ejemplos más representativos en esta categoría es el actual récord mundial sin estrellas de *Super Mario 64* (Nintendo, 1996), donde el jugador utiliza la mecánica de salto y colisiones para ignorar los requisitos necesarios para llegar hasta el jefe final.

- **100 %**: En esta categoría, los jugadores deben de finalizar el videojuego con todos los objetivos cumplidos o, en el caso de que la comunidad lo considere pertinente, solo aquellos que sean necesarios para obtener el máximo requerido. Un ejemplo en esta categoría es el conjunto colaborativo de *speedruns* del juego *Quake* (id Software, 1996) que componen *Quake done Quick*, donde los jugadores superan el juego en la máxima dificultad consiguiendo todos los secretos y eliminando a todos los enemigos de los niveles.

- ***Low %***: Los *speedruns* que entran dentro de este grupo fuerzan a sus jugadores a superar el desafío con la menor cantidad de mejoras u objetos clave posible. Esta condición incluye métodos de transacción en el juego, como dinero, objetos, habilidades y, en ocasiones, niveles de personaje. Por ello, en este tipo de *speedruns* es común que los jugadores aprovechen los errores de diseño del juego o hagan uso de *glitches*. Un ejemplo para esta categoría es el récord mundial actual en el *The Legend of Zelda: Twilight Princess* (Nintendo, 2006) donde el jugador aprovecha un *glitch* que permite atravesar las colisiones del mapa cuando el personaje sostiene una *rupia* [9] como se muestra en la Figura 2.2.

[9]Objeto de los videojuegos de *The Legend of Zelda* que funciona como dinero

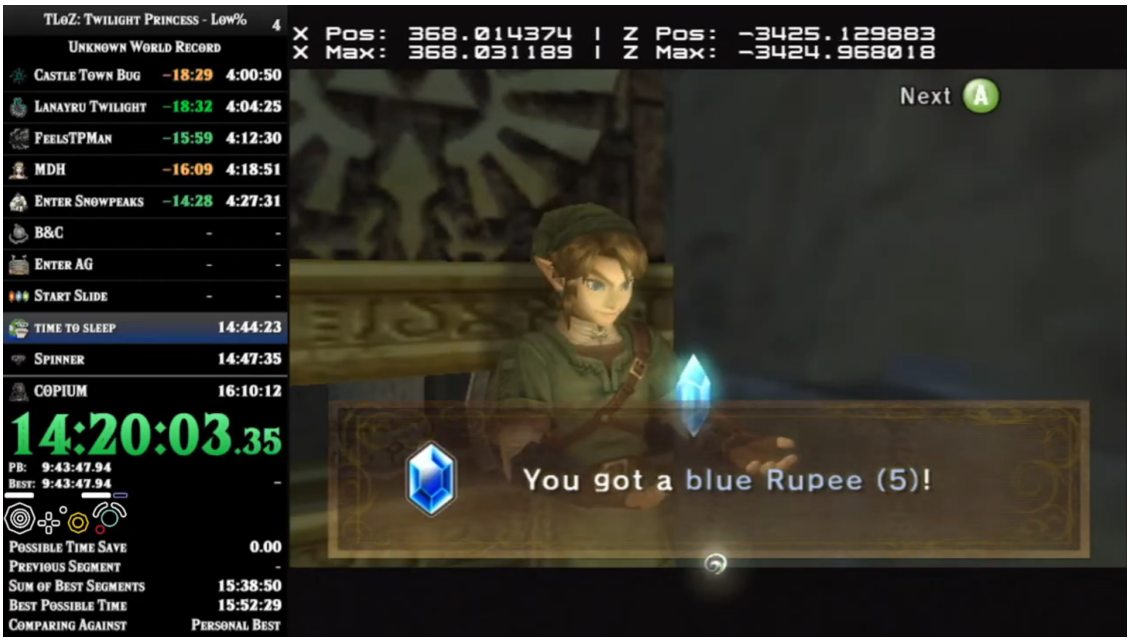

Figura 2.2: Speedrun Low % del juego The Legend of Zelda: Twilight Princess en el cual el jugador aprovecha un glitch de animación al sostener una rupia para poder atravesar los límites del mapa

Sin importar los objetivos impuestos, los *speedruns* también pueden ser agrupados según el flujo de juego que utilizan, siendo estas categorías las siguientes:

- ***Time Attack (TA)***: Los jugadores deben de superar los objetivos establecidos sin tener en cuenta el tiempo utilizado durante los periodos de carga y el reinicio del sistema.
- ***Real Time Attack (RTA)***: A diferencia de lo que sucede en los *speedruns* de la categoría previa, los *speedruns* que pertenecen a este conjunto tienen en cuenta los períodos de carga de pantalla y el tiempo que dura un reinicio del sistema, teniendo los jugadores que buscar soluciones para poder optimizar estos eventos.
- ***Spliced***: En esta modalidad, los jugadores pueden superar los objetivos de manera fragmentada, procurando conseguir la menor cantidad de tiempo global. Por ello, los jugadores pueden repetir varias veces los diferentes desafíos, utilizando en el cómputo global del *speedrun* la cantidad mínima de tiempo empleada en cada sección.
- ***Tool-Assisted Speedrun (TAS)***: Son los considerados *speedruns* perfectos, debido a que son partidas realizadas por *scripts* [10] o

[10]Conjunto de instrucciones escritas en un lenguaje de programación para poder automatizar tareas

emuladores, los cuales ejecutan una secuencia de comandos que, en ocasiones, resulta imposible de replicar para un ser humano.

Esta última categoría, los *speedruns* realizados con *Tool-Assisted Speedrun* son el foco de atención principal de esta propuesta, ya que se busca diseñar una herramienta que facilite la realización de *speedruns* a través de una placa de *Arduino* [11] que contendrá el código necesario.

2.3. Objetivos

Los objetivos que han sido planteados están enfocados en realizar una investigación acerca de la conectividad entre una placa de *Arduino* y un ordenador, con el objetivo de poder ejecutar una herramienta capaz de realizar un *speedrun* automatizado. Todos estos objetivos pueden ser agrupados en dos categorías principales: labor de investigación y desarrollo del proyecto.

En relación con la labor de investigación, los objetivos que han sido establecidos son:

- Realizar una investigación acerca de los *speedruns*, su relación con la industria del videojuego y su papel en la creación de *TASBots*.
- Efectuar un estudio acerca de diferentes librerías de programación, haciendo énfasis en las librerías gráficas desarrolladas con el lenguaje *Python* con el propósito de conseguir las herramientas necesarias para crear un *TASBot*.
- Contrastar cual es la mejor manera de realizar una conexión simple entre una placa de *Arduino* con el propio ordenador, que permita un intercambio de datos correcto entre ambas partes.

En relación con el apartado de desarrollo, los objetivos que han sido planteados son:

- Programar un código funcional con el lenguaje *Python* que permita establecer una conexión con una placa de *Arduino* y realizar un intercambio de datos entre dicha placa y el ordenador.
- Diseñar un algoritmo que pueda lanzar el juego referido en el *TASBot*, en el caso de que no haya sido detectado su proceso, desde la placa de *Arduino* y después ejecutar el propio *TASBot*.

[11] Plataforma de electrónica de código abierto que cuenta con *Software* y *Hardware* fáciles de usar

- Desarrollar un *TASBot* propio que pueda realizar un *speedrun* de un videojuego en 2D con alto componente de aleatoriedad, teniendo una placa de *Arduino* como punto de control para poder ejecutar los *scripts* necesarios para su funcionamiento.
- Una finalizado el proceso de desarrollo, realizar una serie de grabaciones que muestren el ciclo de juego automatizado, que finaliza con el *TASBot* superando el desafío impuesto.

Capítulo 3. Estado del arte

3.1. La industria del videojuego en la actualidad

Desde hace un par de décadas, el mundo de los videojuegos ha avanzado hasta el punto de convertirse en uno de los comercios económicos, culturales y artísticos más importantes, superando en diversas ocasiones al resto de mercados, como podría ser el cine o la música.

Con el comienzo de la era digital, el mercado de los videojuegos logró expandir sus fronteras fuera de los centros recreativos y mejorar la percepción general respecto al medio. Por ello, hoy en día es complicado encontrar a una persona que no juegue o haya jugado a un videojuego.

Gracias a esta situación [4], se han desarrollado comunidades de jugadores, torneos globales, como por ejemplo el torneo mundial de *League of Legends* (Riot Games, 2009), y convenciones, donde es posible interactuar con figuras influyentes del medio, como podrían ser desarrolladores, *streamers*[1] o *speedrunners*.

Aparte del interés que han generado en el mundo del ocio, los videojuegos también se usan como herramientas de investigación en diversos ámbitos, ya que pueden ser utilizados como herramientas de simulación en escenarios hipotéticos donde todas las variables de comportamiento pueden ser modificadas y controladas, además de tener elementos interactivos que fomentan el aprendizaje y la innovación.

Estas simulaciones suelen ser realizadas en laboratorios virtuales que ayudan a la comprensión de eventos científicos complejos por el uso de representaciones visuales o auditivas, sin tener que gastar recursos en el proceso de aprendizaje. Por ejemplo, en el ámbito médico se han dado casos del uso de simulaciones en preparación para una operación quirúrgica delicada y en el campo científico se han empleado simulaciones para calcular el lanzamiento de proyectiles.

[1] Personas que se dedican a transmitir contenido en tiempo real a través de plataformas online

3.2. Simulaciones de lanzamientos de proyectiles en Arduino y Scratch

STEM, cuyas siglas significan *Science, Technology, Engineering and Mathematics*, es el término que reciben los juegos cuyo enfoque permite que los jugadores aprendan del campo científico, tecnológico, matemático o de ingeniería.

En estos últimos años, se han diseñado varios juegos *STEM* centrados en la simulación física de proyectiles, aunque puede destacarse el proyecto que utiliza una placa de *Arduino* y programación por bloques con código en *Scratch*.

Esta simulación [5] se puede conseguir gracias a un *joystick shield*[2] conectado a una placa de *Arduino UNO* que ha sido ensamblada dentro de una caja de madera con *MDF (Medium Density Fiberboard)*. Gracias a este sistema, es posible cambiar el ángulo, la velocidad y la posición inicial del disparo tras realizar el código de programación en *S4A (Scratch for Arduino)*.

El problema de utilizar el lenguaje de programación S4A para hacer el código que utilizará la placa de *Arduino* es que no todos los botones están reconocidos por el lenguaje. Esto implica asignar botones adicionales para poder realizar el lanzamiento de los proyectiles, siendo estos indicados en la Figura 3.1 como 'Digital2' y 'Digital3', mientras que 'Analog1' y 'Analog0' son empleados para determinar la distancia, el ángulo y la velocidad del disparo. Estos elementos están implantados dentro de una placa siguiendo la descripción de cada parte y se muestran en la Figura 3.2.

Figura 3.1: Asignación de los elementos físicos de *input* de una placa de *Arduino UNO* para que puedan ser interpretados por S4A

[2]Placa conectada sobre una tarjeta de desarrollo de tipo Arduino UNO, Leonardo o MEGA que incorpora seis botones y un joystick

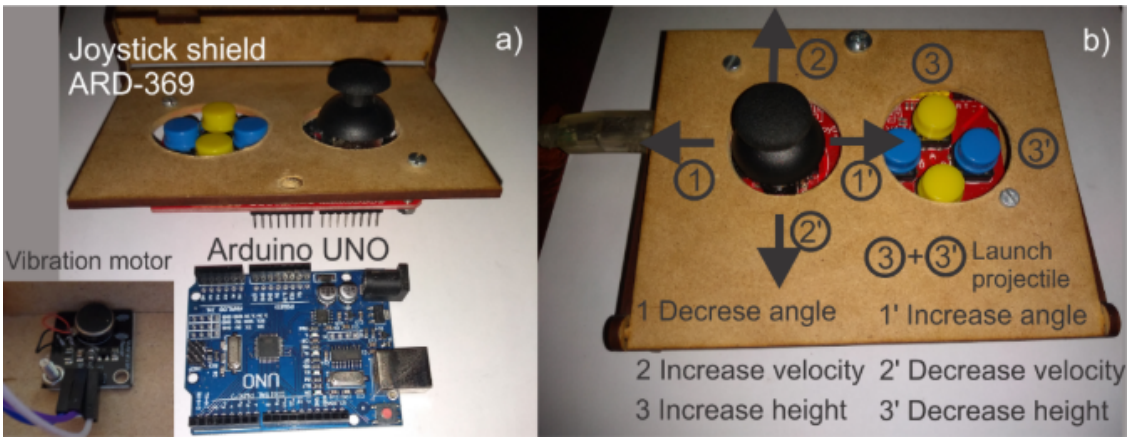

Figura 3.2: El *joystick shield* montado con la descripción de la función de cada parte

Una vez escogidos los parámetros del proyectil con los comandos de *input*, se realiza una simulación del disparo y se realiza un cálculo de trayectoria mediante la siguiente fórmula matemática:

$$y = H + x * tan\theta \ - \frac{g * x^2}{2 * v_0 * cos^2\theta} \tag{3.1}$$

En la Ecuación 3.1 están presentes las siguientes variables: la y representa el resultado de la ecuación de movimiento en el eje vertical, que es igual a H, que indica qué altura se ha obtenido, más x, el cual es el valor de la distancia en el eje horizontal, que está multiplicado por la tangente del ángulo del disparo ($tan\theta$). A esta expresión es necesario restar el resultado de g, que representa la gravedad (siendo en la mayoría de los casos $-9,8m/s^2$), multiplicada por el valor x y dividida entre el doble de v_0 multiplicado por el coseno al cuadrado del ángulo del disparo. [6]

Este proyecto ha proporcionado información para entender cómo funciona la configuración de una placa de *Arduino*, en especial las instrucciones relacionadas con el reconocimiento de procesos y dispositivos externos, necesarios para el desarrollo de un *TASBot*.

3.3. Aplicaciones en procesos de rehabilitación

Los avances tecnológicos y, con ello, los videojuegos han sido importantes para mejorar las condiciones del ámbito médico. Gracias a la creación de simulaciones y dispositivos, se han conseguido tratar desde problemas mentales, como podría ser en el caso de numerosas fobias, hasta físicos, como sucede con la falta de extremidades o la pérdida de movilidad de miembros [7] [8].

Uno de los ejemplos destacables en este campo son las herramientas desarrolladas para mejorar el proceso de rehabilitación de los pacientes que han perdido movilidad en una mano, pudiendo apoyar el proceso de terapia física o sustituirlo, en el caso de la terapia tradicional.

Para ello, se diseñó una interfaz gráfica de usuario *GUI* capaz de registrar gestos con una cámara, con el objetivo de reconocerlos después con aquellos que había guardado en su base de datos, siendo su concepto similar al que puede verse cuando se habla de la **Realidad Virtual Aumentada**.

Una vez establecida esta interfaz, se creó un guante sensorial que facilitaba la realización de gestos, permitiendo a los pacientes tener un ritmo de recuperación superior, aunque la utilidad del dispositivo no terminó ahí, siendo utilizado en instancias posteriores para controlar brazos mecánicos, mejorar el aprendizaje de lenguaje de signos o proporcionar una nueva manera de jugar a los videojuegos, gracias a su control preciso de los movimientos del brazo y los dedos de una mano, otorgando una experiencia mucho más intuitiva y realista en, por ejemplo, juegos de conducción de lo que puede lograr un mando de consola o un teclado.

Para desarrollar este prototipo de guante fueron necesarios, por parte de *Hardware*, un acelerómetro de 3 dimensiones (es decir, que calculaba en los ejes x, y, z) para tener constancia de la orientación del brazo, un módulo *Bluetooth* [4], varios potenciometros deslizantes lineares para localizar la posición de cada dedo y una placa de *Arduino Mega 2560*, mientras que por parte de *Software*, se utilizó *MATLAB*, *Simulink*, *Solidworks* y *Arduino*. Un ejemplo de este prototipo es el mostrado en la Figura 3.3 [9].

La lógica detrás del funcionamiento está en proporcionar los datos con comandos de *input* analógicos a través de la placa de *Arduino* y, con esta información, el modelo del brazo, diseñado con *Solidworks*, podrá rotar y mover sus dígitos, aunque para poder comunicarse con el *Arduino*, fue necesario implementar el modelo de *Simulink*.

[4]Una especificación industrial para redes inalámbricas personales que permite transmitir datos

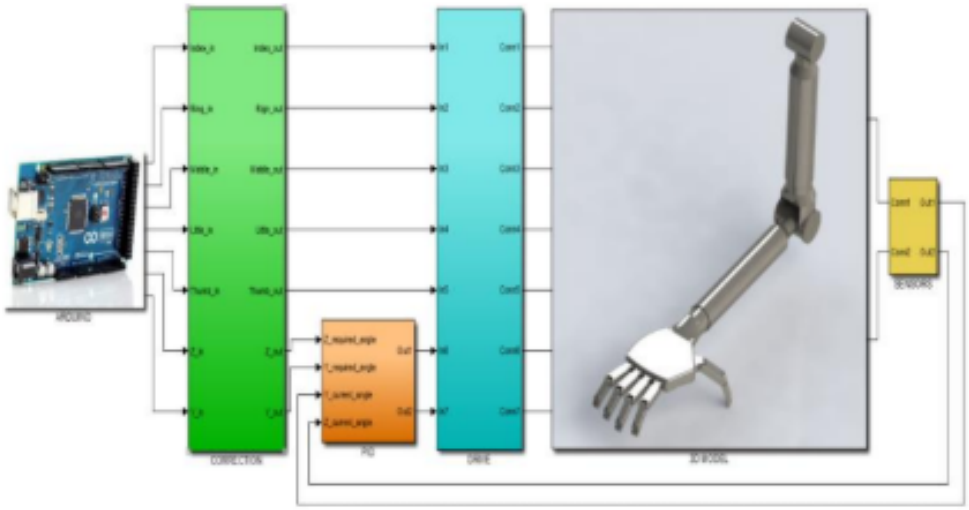

Figura 3.3: El modelo de guante sensorial desarrollado con *Solidworks*, *Arduino* y *Simulink*

3.4. Automatización de labores con Arduino Uno

Otro de los estudios realizados con ayuda de *Arduino* está centrado en los procesos de automatización de trabajos manuales, en especial aquellos relacionados con tareas cotidianas. Dado su bajo costo de implementación y mantenimiento, los sistemas automatizados con ayuda de *Arduino* están ganando popularidad, existiendo numerosos ejemplos en el campo de la botánica, la robótica o la seguridad.

Una de las aplicaciones en botánica se puede encontrar en los sistemas de riego automáticos, que utilizan una placa de *Arduino UNO* para controlar la bomba de agua y los sensores de humedad del suelo, siendo posible configurar el horario de riego y tener control sobre los valores de humedad de manera remota mediante una conexión *Bluetooth* [10].

En el campo de la robótica se pueden hallar usos en el desarrollo de dispositivos capaces de dibujar, pintar y escribir con ayuda de una placa de *Arduino UNO*, que cumple la función de 'cerebro', encargada de procesar el código con los pasos que debe seguir el *robot* mediante un brazo mecánico con motores de paso a paso [11].

Por último, respecto a las aplicaciones en el ámbito de seguridad, existe una implementación de *Arduino UNO* para los sistemas de reconocimiento de placas, que almacena información acerca de las matrículas detectadas con una cámara e identificadas con un procesador de imágenes [12].

Debido a su bajo coste de implementación y su sencillez, *Arduino* es una de las mejores apuestas a la hora de diseñar un proyecto que involucre un

proceso de automatización, motivo por el cual ha sido utilizado para el desarrollo de esta propuesta.

3.5. Emuladores de mandos desarrollados con Arduino

Gracias al desarrollo de diferentes consolas, se han creado numerosos programas, varios de ellos enfocados en el desarrollo de *TASBots*, que permiten emular los controles especializados con herramientas externas, como por ejemplo una placa de *Arduino* [13] [14].

De los ejemplos más destacados, se puede resaltar el proyecto de *ArduinoTAS* desarrollado por el usuario de *GitHub* llamado 'MonsterDruide1' para emular el mando de la *Nintendo Switch* (Nintendo, 2017).

Para lograrlo, el creador ha utilizado una librería *LUFA* e ingeniería inversa de un mando inalámbrico para *Nintendo Switch*, que recibe el nombre *HORIPAD*, con el objetivo de tener una emulación precisa de los controles. Al estar desarrollado en una placa de *Arduino*, carece de varios elementos propios del mando, como son los *joysticks* o los botones del sistema, como *'Capture'* o *'Home'*.

Otro ejemplo de interés es el proyecto desarrollado por el usuario de *GitHub* llamado 'GhostSonic21' con *Arduino* para emular un mando de SNES.

Para conseguirlo, el creador programó los *scripts* necesarios en *Python*, permitiendo que los jugadores pudieran jugar por su cuenta utilizando el teclado o emplear el fichero *TAS* para superar el nivel establecido. El único problema destacable con este programa son los fallos que suceden durante los envíos de información, habiendo numerosos retrasos en el proceso.

El último ejemplo resaltado en esta sección es el emulador de mando de *Sega Saturn* (Sega, 1995) desarrollado con una placa *SparkFun Arduino* por el usuario 'Ryan-Myers'. De todos los ejemplos proporcionados, este en cuestión es el que proporciona la mayor cantidad de problemas a la hora de ser implementado, siendo necesario programar de forma manual y soldar unos elementos adicionales, como un *Mini USB*.

3.6. Herramientas de speedruns enfocadas en Balatro

En el proceso de realización de *speedruns*, se han realizado varios *TASBots* en diferentes lenguajes de programación, que necesitan unos elementos concretos del sistema o externos (como, por ejemplo, *pytesseract* en el caso de un elemento del sistema o una placa de *Arduino* en el caso de los externos).

Utilizando el videojuego *Balatro* (LocalThunk, 2024) como referencia por su componente de partidas con aleatoriedad constante, existen varios ejemplos de herramientas desarrolladas por la comunidad que facilitan el desarrollo de *speedruns*, tanto manuales como realizados con un *TASBot*.

Uno de los ejemplos con mayor reconocimiento es el proyecto desarrollado por el usuario de *GitHub*, 'besteon'. Empleando programación en *LUA* y *Python*, se ha desarrollado una herramienta que, aún sin contener un *TASBot* finalizado, proporciona una base para la creación y el desarrollo de un *bot* funcional y personalizado, que pueda funcionar como *mod* o como *TASBot*.

Otro ejemplo es una calculadora desarrollada por el usuario de *GitHub* llamado 'kleinfreund', que permite obtener el valor de una mano jugada dependiendo de las cartas y los comodines que hayan sido utilizados. En el caso de que haya elementos con aleatoriedad en juego, el programa devolverá el peor, el mejor y el resultado promedio que puede ser obtenido con esa combinación en específico.

Por último, con un enfoque mayor en la realización automática de los *speedruns*, 'raienelliston', un usuario de *GitHub* diseñó un proyecto que utiliza programación con algoritmos desarrollados en *Python* para poder identificar los eventos que suceden en la ventana y hallar el mejor resultado, aunque no tiene implementados todos los casos posibles que pueden ocurrir durante una partida. Este último ejemplo ha sido utilizado como base para estructurar el *TASBot* utilizado en el proyecto.

3.7. La psicología detrás de los speedruns

El aumento de popularidad de los *speedruns* ha provocado un incremento de análisis acerca de la materia, descubriendo los efectos psicológicos, tanto positivos como negativos, que tienen los *speedruns* dentro de la comunidad.

Los efectos negativos suelen estar asociados con el nivel de importancia que los jugadores atribuyen al desarrollo del *speedrun*. Cuanto más relevancia tenga, mayores serán los niveles de ansiedad que pueden adquirir los *speedrunners* durante una *playthrough*, además del exceso de frustración que puede surgir después de perder una partida.

Teniendo en cuenta estos problemas, los *speedrunners* han desarrollado fuertes comunidades basadas en el compañerismo, el deseo de aprendizaje y el sentimiento de pertenencia. Son colectivos pequeños especializados, que disfrutan compitiendo contra otros jugadores, contra ellos mismos o contra el propio juego con el objetivo de conseguir mejores resultados, además de fomentar que otros *speedrunners* mejoren sus propias marcas compartiendo trucos, guías o grabaciones de los intentos que han realizado[15].

Estas comunidades proporcionan un espacio idóneo para que los jugadores experimenten diferentes desafíos, proponiendo rutas alternativas que pueden mejorar el tiempo empleado para terminar el *speedrun*, técnicas nuevas que los *speedrunners* pueden aprender o restricciones que dificulten una *playthrough* normal, habiendo incluso casos donde pueden diseñarse *speedruns* con el objetivo de conseguir un objeto, hablar con un personaje o generar un evento en vez de finalizar un nivel o el juego.

Un ejemplo curioso de un *speedrun* popularizado por las interacciones de la comunidad es el conocido como ***co-op Any% No Controller*** del juego *Barney's Hide & Seek Game* (Realtime Associates, 1993), donde dos jugadores deben superar el juego sin utilizar ningún mando. Otro ejemplo, destacable por su corto plazo de duración para realizarlo, fue un *speedrun* del juego *The Legend of Zelda: Tears of the Kingdom* (Nintendo, 2023) donde la compañía de comida rápida **KFC** (mostrada en la Figura 3.4) propuso a la comunidad de jugadores preparar uno de los platos de comida del juego en la menor cantidad de tiempo posible, planteando una serie de directrices que los *speedrunners* debían seguir para lograrlo.

Figura 3.4: Reglas y pasos que debían de seguir los *speedrunners* para poder participar en la competición de *speedruns* propuesta por KFC

En definitiva, gracias a los *speedruns* se han formado comunidades activas y especializadas dentro del mundo de los videojuegos, donde los jugadores pueden sentirse incluidos y proponer diferentes desafíos que pueden llegar a popularizarse, llegando en ocasiones a ser reconocidos o compartidos por gente ajena a la comunidad.

3.8. Los speedruns como herramienta de localización de vulnerabilidades

Además de plantear un desafío para los jugadores, los *speedruns* han sido utilizados como herramienta de búsqueda de vulnerabilidades en videojuegos [16] [17] y, en numerosos casos, en consolas y ordenadores.

Uno de los mejores ejemplos es el *speedrun* realizado por el *streamer* 'JCog' de *Paper Mario* (Nintendo, 2000), donde utilizó otro título para finalizar la partida en menos de una hora, siendo el título *The Legend of Zelda: Ocarina of Time* (Nintendo, 1998).

Para lograrlo, 'JCog' utilizó el *glitch* conocido como *ACE (Arbitrary Code Execution)* en el juego *The Legend of Zelda*, lo que permitió manipular la memoria interna de la *Nintendo 64* empleada, con el objetivo de lanzar la ejecución del final del juego. Gracias a esta vulnerabilidad, 'JCog' (mostrado en la Figura 3.5) logró mandar el comando encargado de lanzar los créditos finales de *Paper Mario* tras cambiar los cartuchos de los juegos, una vez finalizado el proceso de preparación.

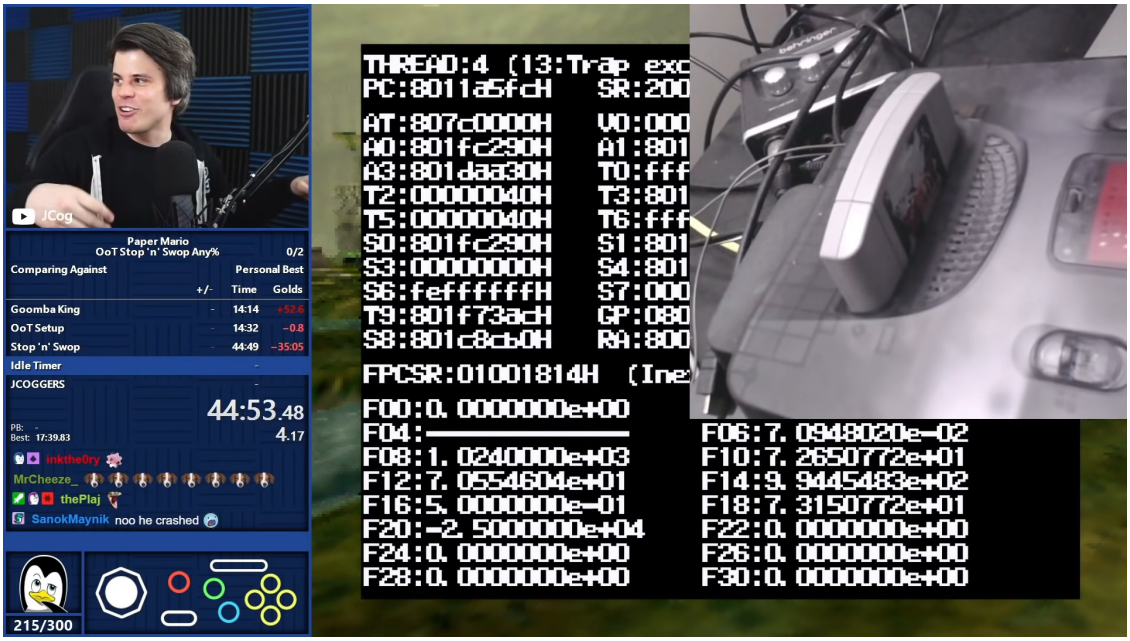

Figura 3.5: 'JCog' cambiando los cartuchos de los juegos después de hacer uso del glitch ACE en mitad del speedrun

Otro caso donde los *speedruns* han sido clave para encontrar los problemas que puede tener un programa es el que ocurre con *Dragon View* (Nintendo, 1994). Dentro del juego existe un contador que aumenta después de acabar con un *boss*[2] pero que solo puede reiniciarse cuando el jugador apaga la consola. Esto provoca que, después de derrotar varios *bosses*, la memoria del juego comience a quedar saturada, pudiendo ejecutar la secuencia de los créditos finales, aunque no es un evento que suceda siempre. El *streamer* 'PJ DiCesare' ha hecho numerosas pruebas de ingeniería inversa con el objetivo de buscar una explicación, publicando un speedrun con el proceso comentado paso a paso.

Aunque no sea siempre el foco principal de la modalidad, los *speedruns* siempre han utilizado todas las herramientas que tenían disponibles para finalizar un desafío en la menor cantidad de tiempo posible, siendo por

[2]Enemigo en un videojuego que suele implicar una dificultad superior

ello uno de los mejores métodos de la industria para poder localizar vulnerabilidades tanto en videojuegos como en consolas.

3.9. Los speedruns y su relación con redes neuronales

Una red neuronal, en programación, es un modelo de aprendizaje automático que imita el funcionamiento de un cerebro, con el objetivo de que un programa pueda tomar decisiones después de identificar un problema y proponer opciones [18].

En relación a los videojuegos, las redes neuronales son utilizadas para estructurar, por ejemplo, el comportamiento de los NPC [3], los enemigos o los objetos del escenario [19]. En el ámbito de los *speedruns*, por otro lado, las redes neuronales son empleadas para crear los ficheros TAS.

Sin importar los elementos que haya dentro de un videojuego, es posible realizar un *speedrun* con *TASBot*. Esto es debido a que, durante la preparación de la red neuronal, el *TASBot* irá estableciendo 'árboles de toma de decisiones' dependiendo de la reacción obtenida tras realizar una pulsación concreta de botón, comenzando a trazar rutas cada vez más eficientes [20], como se puede apreciar en la Figura 3.6.

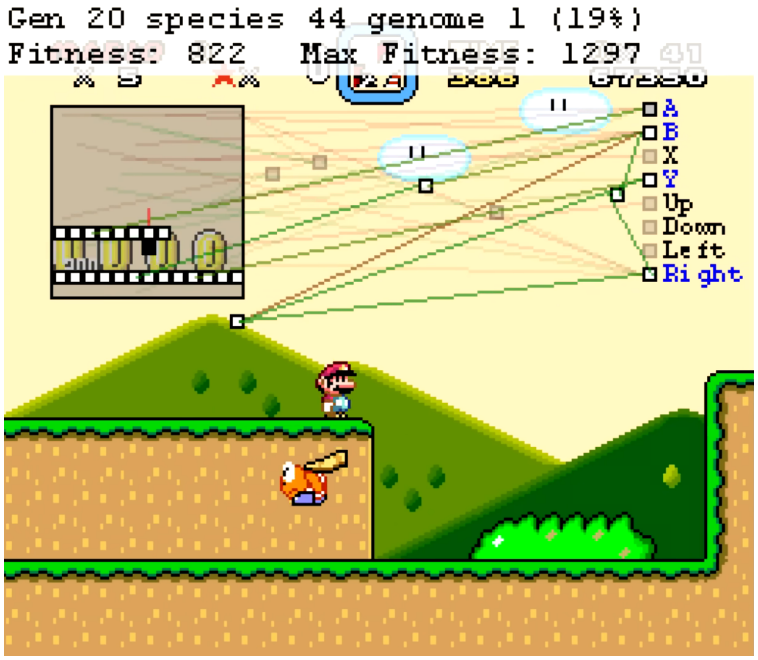

Figura 3.6: Mar/O, un programa que utiliza redes neuronales y algoritmos genéticos para realizar un speedrun del Super Mario World

[3] Personajes no controlados por el jugador en un videojuego

En los casos donde haya elementos RNG, como se mostrará en la Sección 4.2, sigue siendo posible realizar *speedruns* con ficheros TAS, debido a que los componentes que puedan involucrar aleatoriedad, como podrían ser los objetos que pueden aparecer en el escenario, el tipo de criaturas que merodean por el mapa o el incremento de habilidades recibidas tras subir de nivel, están predefinidos en el código. Al requerir unas condiciones concretas para poder generarse, el entrenamiento de una red neuronal para un *TASBot* tendrá en cuenta estos eventos, pudiendo ignorarlos o tenerlos en cuenta, llegando a forzar las mecánicas para obtener los resultados esperados.

En conclusión, el entrenamiento de una red neuronal durante la preparación de un *speedrun* con *TASBot* genera un método interactivo y sencillo para entender conceptos del aprendizaje automático que, de otra manera, resultarían difíciles de visualizar. Respecto al desarrollo del proyecto, se ha empleado la información obtenida sobre las redes neuronales para intentar que el *TASBot* dé prioridad a unas jugadas por encima de otras, modificando su comportamiento conforme avanza la partida.

Capítulo 4. Tecnologías

4.1. TASBot

El archivo utilizado para realizar este *speedrun* fue un *Tool-Assisted Speedrun* o *Tool-Assisted Superplay*, que funciona guardando *bytes*[1] de memoria que se consideran como las pulsaciones de botón realizadas antes de un envío de señal al controlador, siendo 1 si el botón ha sido presionado y 0 si no. El problema con este sistema radica en la diferencia de tiempos entre señales, siendo estas recibidas en el transcurso de nanosegundos pero enviadas tras unos milisegundos.

Un *TAS* o *Tool-Assisted Speedrun* es *speedrun* realizado con ayuda de un emulador *frame* a *frame*, que puede llegar a ser replicado por un jugador, aunque dicha persona requeriría una gran capacidad de memorización y reflejos que, en ocasiones, son imposibles de replicar por un ser humano. Por este motivo, los *speedruns* hechos con un fichero *TAS* son considerados *playtroughs* perfectos.

Crear un archivo *TAS* se puede hacer de varias formas, pero en todas las opciones, el objetivo final es encontrar el conjunto óptimo de *inputs* que permita completar el *speedrun* con el menor tiempo posible. La manera estándar de usarlo es grabando en un archivo los comandos de botón paso a paso mientras se juega en un emulador, habiendo una serie de herramientas que facilitan la creación del *TAS* y la realización del *speedrun*, siendo los siguientes nombres los más reconocibles:

- ***Savestates***: Un *savestate* es el estado guardado con anterioridad del emulador, lo que permite volver a cargar una partida en caso de haber cometido un fallo o que se haya producido un error.

  Además, permite decidir cúando comienza una grabación del *TAS Video*, pudiendo empezar desde un momento específico indicado por el jugador o desde el momento en el que se ha iniciado el *TAS*.

- ***Slow-down***: Es una herramienta que permite ralentizar el videojuego para facilitar la grabación de partidas y la introducción de comandos de *input*.

[1] Unidad de información compuesta generalmente de ocho bits

- ***Memory Watch***: Permite ver los valores internos de un videojuego que posea varios atributos, como los ángulos de movimiento, las posiciones de los elementos del juego o el propio factor RNG (*Random Number Generator*), que se mostrará en la Sección 4.2, pudiendo visualizarlo *frame* a *frame*.

- ***Glitches***: Un *glitch* es un comportamiento inesperado por parte de los desarrolladores durante el proceso de programación que produce resultados no intencionados.

  Existen *glitches* de diversos tipos, destacando aquellos donde los jugadores logran salir de los bordes establecidos del mapa, duplicar objetos, saltar diálogos o, en determinadas situaciones, ejecutar código de manera arbitraria, siendo este caso conocido como *ACE (Arbitrary Code Execution)*, el cual es utilizado para introducir comandos personalizados en una consola.

El uso de las herramientas mencionadas anteriormente es beneficioso para, por ejemplo, descubrir nuevas técnicas que permitan reducir los costes de tiempo durante un *speedrun* sin el uso de un fichero *TAS*. A pesar de lo indicado, un archivo *TAS* suele contar con la carencia del factor humano que facilita la falta de errores durante el desarrollo de un *speedrun*.

Para poder desarrollar un archivo *TAS*, es necesario encontrar el conjunto de comandos de *input* que permitan superar el desafío impuesto, pudiendo ser estos difíciles de obtener por las capacidades del ser humano. No existe ningún límite respecto a las herramientas que deben usarse para crear el fichero *TAS*, pero el listado de los botones pulsados debe tener en cuenta las limitaciones de ejecución del *script*, para poder conseguir completar el objetivo establecido.

El método básico de creación de un listado de comandos es grabar la secuencia de pulsaciones de botón que realiza un jugador en un emulador, repitiendo el proceso con diferentes posibilidades hasta obtener el mejor resultado. Además de grabar el procedimiento, pueden usarse diversas herramientas que faciliten esto, como podrían ser:

- ***Precise timer***: Un contador *frame*

- ***Re-recording***: Una herramienta que permite deshacer acciones o pulsaciones de botón introducidas por error, facilitando las pruebas de ensayo y error en determinados momentos de una partida.

- ***Hex-editing***: Un editor que permite copiar y modificar de manera manual los comandos de *input* introducidos, pudiendo corregir fallos o perfeccionando el transcurso de una partida.
- ***Autofire***: Una herramienta que facilita las pulsaciones automáticas de un botón establecido como determinado, pudiendo ser el comando para guardar partida, saltar, atacar, etcétera... dependiendo esto de las mecánicas del juego del cual esté haciéndose un *speedrun*.

4.2. Los algoritmos RNG

A la hora de hablar de *speedrun* suele ser común enlazarlo con el concepto de *Randon Number Generator*[2], conocido popularmente como *RNG*. El *RNG* es utilizado para crear una sensación de aleatoriedad en determinados aspectos de un videojuego, como pueden ser patrones de comportamiento en enemigos, recompensas, salas generadas, etcétera.

El *RNG* funciona mediante un conjunto de normas que cambian dependiendo del videojuego y es, gracias a estas reglas, por lo que muchos *speedrunners* lo emplean como factor determinante a la hora de realizar una partida con un fichero *TAS*.

A pesar de servir en todos los casos para el mismo propósito, es posible dividir los videojuegos que utilizan *RNG* en diferentes categorías, dependiendo: de cuándo emplean elementos aleatorios, pudiendo generar números nuevos después de que sucedan eventos importantes, de manera frecuente por motivos superficiales o con cada *frame*; o de qué tecnología emplea el algoritmo, siendo estos:

- ***TRNG (True-Random Number Generator)***: Son algoritmos *RNG* que generan números aleatorios a raíz de los elementos que pueden ser observados en un entorno natural, como por ejemplo el ruido estático o un cambio de temperatura.
- ***PRNG (Pseudo-Random Number Generator)***: Estos generadores utilizan fórmulas matemáticas para obtener secuencias numéricas que parecen aleatorias, pero sin llegar a serlo. Funcionan utilizando un número inicial, llamado 'semilla' o *seed*, a partir del cual se genera el resto de la cadena numérica.

Debido a que utilizan elementos físicos reales para sus cálculos, los modelos *TRNG* no suelen ser empleados para el desarrollo de videojuegos,

[2]Definición:https://gaminglabs.com/glossary-of-terms/random-number-generator/

en los cuales es común que los desarrolladores busquen tener control de los elementos generados. Esto es posible usando algoritmos *PRNG*, donde los desarrolladores pueden manipular la *seed* para evitar que genere los mismos números cuando el jugador inicia una partida, que puedan eliminarse números de la secuencia o que solo puedan obtenerse unos resultados específicos [21].

Un elemento *seed* puede ser definido por el archivo de guardado empleado, los eventos de juego que han sucedido, el reloj interno del dispositivo empleado o, en algunos casos, los comandos de *input* utilizados en ese momento.

A la hora de realizar un *speedrun* que requiera manipular el algoritmo *RNG*, los *speedrunners* deben tener en cuenta el método empleado por el videojuego para generar las nuevas secuencias de números. Por ejemplo, si son definidos con el *input*, solo necesitan modificar el método con el cual introducen los comandos. Si, por otro lado, son generados cada *frame*, entonces tendrán que provocar eventos de *delay*[3] que permitan manipular los procesos del ciclo de juego. Por último, si los números son generados después de que sucedan unos eventos determinados, el proceso de manipulación será complicado, teniendo los *speedrunners* que forzar en ocasiones los límites del juego para lograrlo.

Dentro de los algoritmos que pertenecen a la categoría *PRNG*, cabe destacar el *LCG (Linear Congruential Generator)*, cuyo funcionamiento es el siguiente:

1. Primero, se multiplica el valor seleccionado (el número empleado puede ser la *seed*) por una constante que realizaría una escala ´a‘.
2. En caso de que haya, añadimos un incremento ´c‘.
3. Por último, se realiza una división entre el módulo ´m‘, siendo el resto congruente.

Siguiendo los pasos, la fórmula matemática de un algoritmo *LCG* es:

$$X[n+1] = \frac{(a * X[n] + c)}{|m|} \tag{4.1}$$

En la Ecuación 4.1 están presentes las siguientes variables: la a representa un número entero que funciona como multiplicador,

[3]Pausa ocurrida durante la ejecución de un programa, en ocasiones a propósito

multiplicado por *X[n]*, el cual es el valor actual de la secuencia. A este resultado se suma *c*, que representa la constante de incremento y, una vez que se ha realizado la suma, se divide la solución entre el módulo de *m*, que define el rango de los valores posibles. Por último, *X[n + 1]* representa el siguiente valor aleatorio generado.

4.3. Proyecto EPO y Cronus Zen

Existen herramientas que permiten a los jugadores superar los desafíos con mayor facilidad o mejorar su rendimiento durante una partida, gracias a una serie de cambios internos o una ejecución de *scripts* que facilita el apuntado de objetivos con el ratón.

Dentro de los ejemplos de *Hardware* enfocados en tratar los problemas del apuntado del ratón, destacan **Proyecto EPO** y **Cronus Zen**.

Proyecto EPO es un dispositivo que modifica elementos a nivel de *Hardware* para facilitar el proceso de apuntado en videojuegos, siendo indetectable gracias a que no requiere ninguna ejecución de *software* para su funcionamiento, aunque requiere tener una conexión directa con el ordenador o la consola para poder usarse.

El funcionamiento del dispositivo **Proyecto EPO** es el siguiente:

1. En primer lugar, una herramienta llamada ***DMA Device*** lee la memoria del sistema sin necesidad de ejecutar programas de *software* dentro del ordenador.
2. A continuación, con ayuda del dispositivo ***Raspberry Pi***, las coordenadas de los datos leídos son enviadas al ***EPO Aim Device***.
3. En el último paso, la herramienta *EPO Aim Device* se conecta de forma física en el ordenador, asumiendo el papel de ratón y recibiendo por ello los datos relacionados con el estado y el posicionamiento del puntero, los cuales modifica para después volver a enviarlos de vuelta.

Una vez conectado el dispositivo *EPO Aim Device*, el jugador dispondría de apuntado asistido o automático, dependiendo de las características establecidas, que podría tener como una función pasiva o, en el caso contrario, disponer de la facultad para desactivar o activar el *aimbot*.

Por otro lado, **Cronus Zen** es un dispositivo *Hardware* de pequeño tamaño que puede ser conectado al mando de un jugador o a uno de los

puertos de un ordenador o consola, con el objetivo de proporcionar a los jugadores una herramienta que proporcione una ventaja en el transcurso de una partida, modificando en el proceso el equipamiento del videojuego.

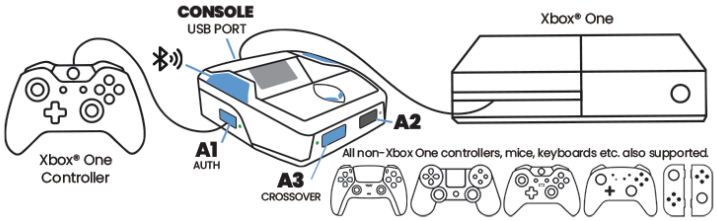

Figura 4.1: Imagen de ejemplo de uso de Cronus Zen para una *Xbox One*, mostrando su disponibilidad para funcionar y conectar varios mandos a la consola de dispositivos diferentes a la propia *XBox*

Tras conectar el dispositivo, los jugadores pueden añadir *scripts* que proporcionen beneficios durante una *playthrough*, como por ejemplo un sistema de apuntado automático en un videojuego de disparos o una baliza de localización que identifique los objetivos en un mapa. Para lograrlo, **Cronus Zen** imita el comportamiento de un botón o los *joysticks* de un mando a través de unos ficheros que pueden ser conectados a cualquier consola que lo permita, siendo conocido dicho comportamiento de manera coloquial como 'emulación de control'.

4.4. OpenCV

OpenCV[4] es una librería de código abierto utilizada para el reconocimiento de los datos gráficos que son generados en la pantalla. Esta librería está desarrollada en *C++*, aunque puede ser utilizada en *Python*, *Java*, *Javascript*, *Matlab* y *Octave*.

Esta librería permite realizar numerosas funciones relacionadas con el apartado de detección gráfica, siendo empleada por ello para labores de reconocimiento facial, detección de objetos, seguimiento del movimiento de un elemento en específico, realidad aumentada, etcétera.

En el ámbito de los *speedruns*, es una herramienta bastante útil para el desarrollo de *TASBots*, gracias a que pueden visualizar los cambios en el estado de juego en tiempo real, interpretar dichas variaciones y transmitir la información obtenida para que los algoritmos puedan reaccionar a los datos recibidos.

[4]Enlace a la librería:https://opencv.org/

Para ello, *OpenCV* cuenta con métodos para realizar capturas de pantalla, localizar determinados objetos en un escenario, identificar si ha habido modificaciones respecto a instancias previas de la escena o seleccionar puntos concretos de la ventana, entre muchos ejemplos.

4.5. PyAutoGui

PyAutoGui[5] es una librería fácil de utilizar que ha sido programada en *Python* para realizar labores de automatización de tareas, destacando aquellas relacionadas con el desplazamiento del cursor, la lectura de comandos introducidos por teclado o la pulsación de botones, sin importar que puedan aparecer en otra ventana.

Entre las funcionalidades proporcionadas por la herramienta, una de las más destacadas permite mover el cursor del ratón a una posición establecida por los atributos 'X' e 'Y' y seleccionar de manera automática sobre dichas coordenadas de la ventana, pudiendo llegar a interactuar con un elemento presente en pantalla. Otra de las funciones más relevantes permite modificar el tamaño y el posicionamiento de las ventanas que están abiertas en pantalla.

Por último, es importante resaltar que *PyAutoGui* proporciona instrucciones para mostrar mensajes por pantalla en ventanas secundarias de un solo uso.

4.6. SubProcess

Para poder crear nuevos procesos en el lenguaje *Python* existe una herramienta llamada *Subprocess*[6], un módulo de programación que permite que sus usuarios puedan ejecutar comandos externos a aquellos que han sido declarados en el *script*. Esto concede a los clientes capacidad de interactuar con los procesos como si lo hicieran desde una terminal o de ejecutarlos como si fuesen fragmentos de código, además de ser capaces de obtener los datos referentes a la entrada y salida de *input*.

Una de las opciones destacadas que pueden ser utilizadas por esta herramienta es la ejecución de procesos secundarios en paralelo, permitiendo que puedan pasarse rutas de ficheros como atributos en funciones con el objetivo de poder abrir varios programas diferentes de manera simultánea. Otra de las funcionalidades que proporciona está relacionada con el tratamiento de las excepciones, pudiendo proveer

[5] Acceso a la librería: https://pyautogui.readthedocs.io/en/latest/
[6] Acceso a la herramienta: https://docs.python.org/3/library/subprocess.html

mensajes por terminal acerca del error que ha sucedido sin interrumpir la ejecución del programa, siempre que no sea un bloque de código indispensable.

4.7. PyTesseract

PyTesseract[7] es una herramienta para *Python* de *OCR (Optical Character Recognition)* encargada de reconocer y leer los textos incorporados en las imágenes, compatible con bibliotecas gráficas como *Leptonica* y *Pillow*, incluyendo las imágenes con formato *JPEG*, *PNG*, *GIF*, *BMP* y *TIFF*.

Para poder realizar sus funciones, *PyTesseract* procesa una imagen con el objetivo de mejorar su calidad. Después, examina la organización de la página para determinar si en el archivo existen bloques de texto y, mediante patrones de reconocimiento, busca similitudes en áreas segmentadas de la imagen.

Una vez finalizado el proceso de preparación de los datos, *PyTesseract* convierte los bloques de texto y los elementos gráficos de una imagen escaneada en un mapa de *bits*. Una vez obtenido este mapa, es posible trabajar con el resultado en *OpenCV*.

Los caracteres reconocidos por parte de *Tesseract* se consiguen gracias a una combinación de algoritmos diseñados mediante *Machine Learning* a través del procesamiento de imágenes convencionales. Debido a la diferencia de idiomas existentes, muchos de los cuales cuentan con caracteres propios, *PyTesseract* cuenta con procesos de corrección lingüística, que mejoran los resultados obtenidos.

Además del procesamiento de imágenes o la revisión de texto, *PyTesseract* proporciona operaciones que permiten convertir el contenido de una imagen en texto o en una señal que pueda modificar el funcionamiento del código, guardar los caracteres que han sido reconocidos y analizar su posicionamiento o su orientación.

4.8. Arduino

Arduino[8] es una plataforma de código abierto utilizada para la creación de sistemas electrónicos, teniendo como base diversos elementos de *Hardware* y una programación de *Software* libre, flexible y sencilla de utilizar. Debido a estas características, *Arduino* suele ser empleado para el desarrollo y la creación de dispositivos con diferentes usos, como puede

[7] Enlace a la herramienta:https://pypi.org/project/pytesseract/
[8] Enlace a la plataforma:https://www.arduino.cc/

ser el encargado de realizar el cambio de colores de un semáforo o un *robot* capaz de detectar y seguir una línea marcada en el suelo.

Una placa de *Arduino* está basada en un microcontrolador[9], unos conectores de salida y entrada que tienen la funcionalidad de una luz *LED*, una fuente de alimentación y una plataforma de desarrollo llamada *Arduino IDE* donde puede ser programado y ejecutado el código que necesita el programa para funcionar, siendo dicho código cargado después dentro de la propia placa de *Arduino*.

En el desarrollo del proyecto se optó por emplear una placa *Arduino UNO R3 WiFi ATmega328P+ESP8266* combinada con un microcontrolador *ATmega328P* y un módulo *Wi-Fi ESP8266*. Elegimos esta placa debido a que nos permitía hacer una conexión mediante USB pero también dejaba abierta la posibilidad de llegar hacerlo mediante una conexion remota mediante *Wi-Fi*.

Dentro de los componentes incluidos en la placa, pueden hallarse:

1. ***Microcontrolador Principal (ATmega328P);*** Este *chip* puede encontrarse también dentro de un *Arduino UNO R3* estándar, siendo una de las mejores opciones para ejecutar tareas de control en tiempo real, lectura de sensores y manejo de los conectores.

2. ***Microcontrolador Wi-Fi (ESP8266EX):*** Es un potente microprocesador con *Wi-Fi* integrado, encargado de dirigir toda la conectividad inalámbrica.

3. ***Convertidor USB-TTL CH340G***: Este *chip* funciona como un puente entre uno de los puertos USB del ordenador y los conectores de la placa, siendo estos los conectores TXD, encargado de la transmisión de datos, y RXD, encargado de la recepción de información.

4. ***Interruptores DIP (Dual In-line Package)***: Es un bloque de ocho interruptores que permiten redirigir las conexiones, aunque primero debe haber sido seleccionado el microcontrolador que será conectado con el *chip* CH340G (y, por consecuente, con el puerto USB) o establecer que los dos microcontroladores serán conectados entre sí.

[9]Chip considerado como el bloque principal de una placa de *Arduino*

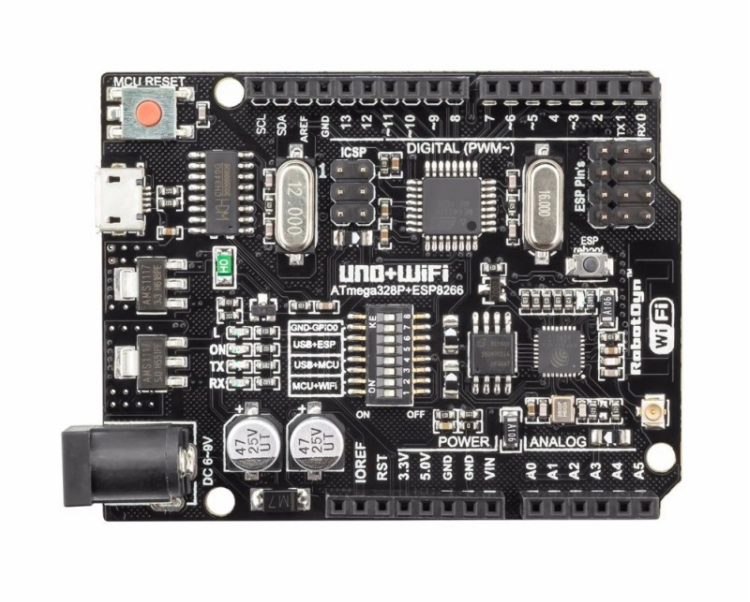

Figura 4.2: Imagen de la propia placa de *Arduino UNO R3 WiFi ATmega328P+ESP8266*

Para poder hacer funcionar una placa de *Arduino*, es necesario activar los interruptores SW3 y SW4, mientras que todos los demás permanecen desactivados. Una vez realizada esta configuración, la comunicación puede ser establecida después de conectar TXD0 y RXD0 con TXD y RXD gracias a los interruptores DIP.

Capítulo 5. Diseño de la solución

5.1. Conexión con Arduino

La solución inicial comienza con un código de *Python* que inicia un bucle de búsqueda por todos los puertos del ordenador, que finaliza cuando recibe una señal. Tras especificar el puerto que recibe la señal, se lanzan una serie de instrucciones para confirmar que la conexión permite realizar el envío correcto de *bits* a través del puerto por el que ha sido conectado el *Arduino*.

Estas conexiones realizadas con la placa de *Arduino UNO* han sido ejecutadas en un *script* de *Python* gracias a la librería *PySerial*, que funciona como una extensión de *Python* para *Windows* y que permite adquirir información a través de los puertos del ordenador y los procesos de configuración.

Una vez realizado el intercambio de mensajes entre la placa de *Arduino* y el ordenador, el código de *Python* mostrará un mensaje de confirmación por la terminal que comenzará el proceso de lanzamiento del *TASBot* diseñado y del videojuego que se quiere superar, siendo en este caso el juego *Balatro* (LocalThunk, 2024).

En la Figura 5.1 se pueden apreciar los pasos realizados por el código gracias a los mensajes mostrados en la terminal de *Python*. Como se puede apreciar, una vez iniciado el *script*, este espera hasta recibir una señal correspondiente con una placa de *Arduino* para continuar ejecutando el resto de las instrucciones. Una vez localizada la señal, se comprueba, gracias al contenido de la señal que ha sido enviada, que la placa de *Arduino* contiene los *scripts* necesarios para realizar el *speedrun*. En el caso de que todos los pasos hayan salido bien, se ejecuta el videojuego y, tras un pequeño período de pausa para que el juego pueda terminar de cargar, se lanza el *TASBot*. 5.2

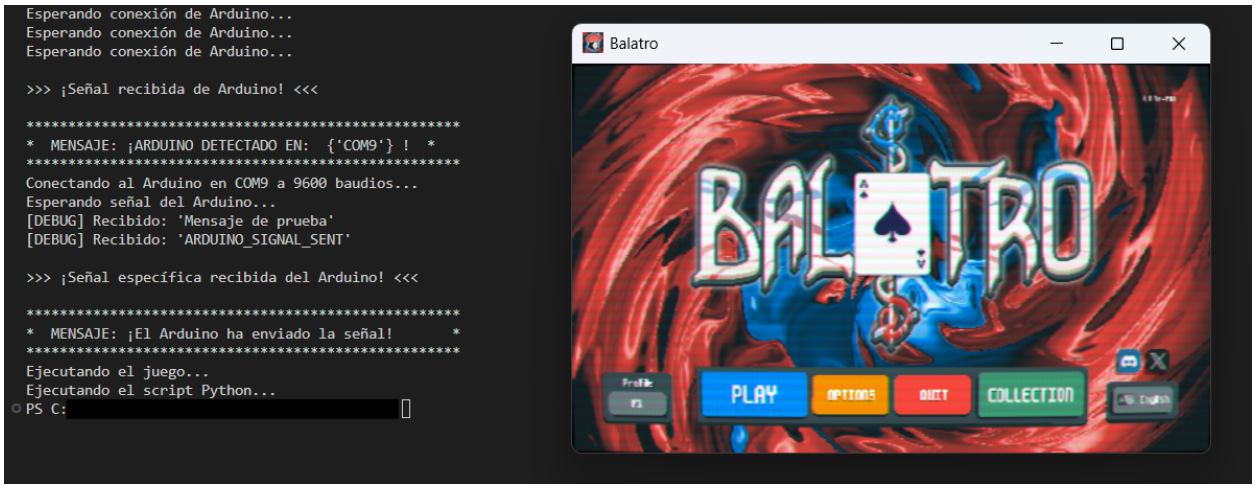

Figura 5.1: Una terminal de *Python* que muestra por mensaje los pasos realizados por el código y la ventana de *Balatro* que acaba de ser lanzada

5.2. Diferencia y cambio entre escenas

Antes de hablar del desarrollo del *TASBot*, es necesario explicar el diseño implementado para obtener un bucle de juego, el cual está compuesto por solo cuatro escenas que pueden apreciarse en el diagrama de la Figura 5.2: el menú principal (representado con *menu scene*), la selección de ciegas (representada con *blind scene*), la tienda (representada con *shop scene*) y la zona de juego (representada con *hand scene*).

Estas 'escenas', que en realidad son cuatro funciones guardadas en un módulo de *Python* llamado *scenes.py*, fueron implementadas después de comprobar que había problemas para conseguir que se lanzaran instrucciones específicas de lógica, pudiendo ejecutarse métodos relacionados con la escena de la tienda durante los eventos de selección de ciegas. Gracias a estas cuatro funciones, las 'escenas' están encargadas de gestionar su propio sistema de organización y, a su vez, estos métodos están estructurados dentro de una función encargada de supervisar el ciclo de juego.

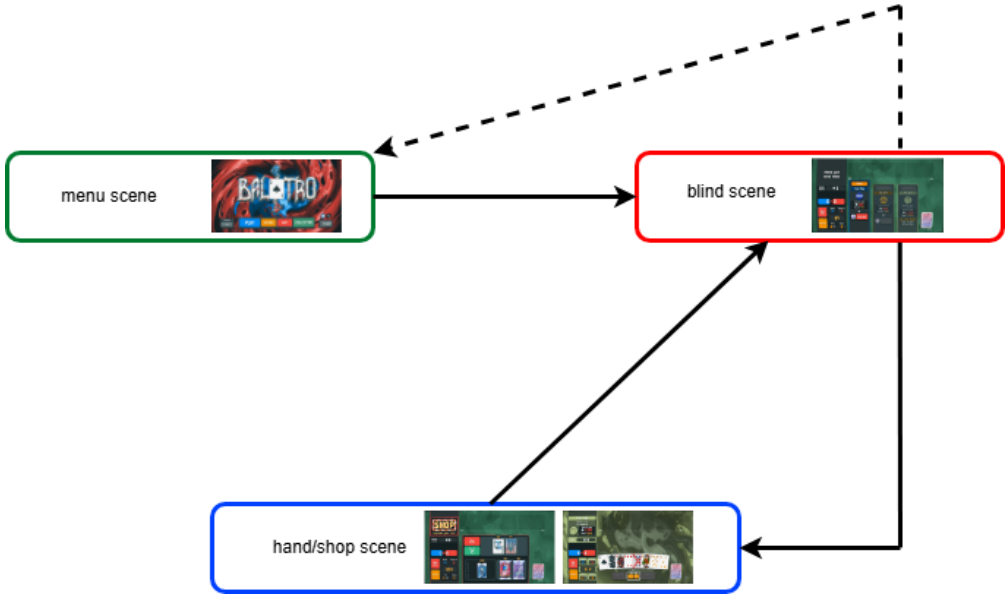

Figura 5.2: Diagrama que muestra el ciclo de juego, marcando con lineas discontinuas las rutas 'opcionales' en el caso de volver al menú

Usando como referencia la Figura 5.2, este ciclo de juego comienza en la escena del menú principal, representada con el color verde. Una vez situado en esa escena, el *TASBot* iniciará una partida pulsando el botón que abrirá la ventana de selección de ciega, ilustrada en la Figura con el color rojo. Desde ahí, el *TASBot* escogerá el nivel que quiere jugar y empezará una puja, cambiando a la escena de juego que ha sido representada con el color azul en el diagrama. En el caso de ganar, se iniciará la escena de tienda (de color azul también), desde donde el *TASBot* entrará, de nuevo, en la pantalla de selección de ciega, seguido de otra escena de juego y así hasta que pierda la partida, momento en el cual regresará al menú principal.

5.3. Localización de los elementos en pantalla

Respecto a la creación de una herramienta de asistencia en videojuegos, es importante diferenciar entre los videojuegos con elementos fijos, cuyo comportamiento no sufre ninguna clase de cambio dependiendo de las acciones del jugador, siendo un ejemplo el *Super Mario 64* (Nintendo, 1996), donde todos los enemigos y los obstáculos tienen un comportamiento predefinido, y aquellos donde los elementos son manipulados por interacciones con el jugador o por mecánicas de aleatoriedad, como por ejemplo encontramos en el *Hades* (Supergiant, 2020), donde todas las partidas están formadas por un mapa procedural y las habilidades empleadas tienen porcentajes aleatorios de aparición, que pueden variar

dependiendo de las acciones del jugador.

Para poder diseñar un *TASBot* es posible guardar de forma manual los comandos en un *script*, pero este método solo es funcional cuando el videojuego carece de elementos de aleatoriedad que no puedan ser previstos o calculados con anterioridad. Para evitar esta situación, en el proceso de desarrollo de un *TASBot* es común implementar métodos que permitan reconocer y procesar los elementos que son mostrados por pantalla.

El método empleado normalmente para programar estos métodos suele utilizar *OpenCV*,una vez realizada la inclusión de la librería, el siguiente paso requiere configurar un método de lectura general, que busque los elementos requeridos dentro de una imagen mostrada por la pantalla.

Para poder localizar los componentes en cuestión, *OpenCV* requiere archivos de imagen que muestren solo los elementos indicados. Con ayuda de las imágenes, el método puede hallar las coordenadas requeridas para el funcionamiento del programa y, dependiendo del comportamiento que haya sido programado en respuesta, realizar unas acciones u otras, que pueden variar desde seleccionar hasta desplazar objetos de una posición a otra.

Además de las imágenes de los elementos que necesitan ser identificados del juego, es importante tener un método que permita que el programa extraiga el estado general de la pantalla. Para lograrlo, una sección del código debe ser destinada a la captura de ventana e identificación del estado de juego, habiendo una distinción entre los componentes que pueden ser encontrados en una pantalla o en otra. Por ejemplo, un botón que sea utilizado para 'comenzar una partida' tendrá mayor probabilidad de aparecer en el menú de inicio que durante uno de los niveles del juego.

A la hora de desarrollar el proyecto, hubo problemas con la implementación de estos métodos de identificación, debido a que las imágenes utilizadas para extraer información de la captura eran bastante similares entre ellas. Este contratiempo logró resolverse tras quitar los filtros de pantalla del propio juego, en especial el *CTR Shader*.

5.4. Conexión con el juego

Para realizar la conexión entre una placa de *Arduino UNO* y un videojuego ejecutado en el ordenador, se ha optado por utilizar *pyWin32*,

una librería que funciona como extensión de *Python* para *Windows*.

Con esta librería, se pueden identificar los procesos que están siendo ejecutados en el ordenador, permitiendo detectar ventanas específicas. Al tener que realizar una conexión con un elemento concreto, es necesario que las instrucciones enfocadas en la captura de pantalla estén situadas en el inicio del código, indicando en el proceso el nombre de la ventana en cuestión.

En el caso de localizar el proceso, es posible modificar el tamaño de la ventana con el objetivo de ajustar sus dimensiones a la pantalla. Tras realizar las modificaciones necesarias, se crea un *bitmap* [1] con el ancho y alto de la imagen, mediante una instrucción llamada *Bit Block Transfer*, cuya función es copiar los píxeles entre dos dispositivos, marcando el punto inicial, el tamaño que debe ser almacenado, la ubicación donde serán capturados los píxeles (que en esta ocasión será la ventana indicada) y especificando el modo de copia.

Tras guardar los datos en el *bitmap*, se obtienen los bits pertinentes que serán convertidos en un *array*[2] plano de ocho *bits*. Este procedimiento puede provocar una ralentización del sistema, evento que puede ser evitado con una liberación de recursos, tanto del *bitmap* como de los accesos a ventana como se comento en el 4.3.5.3

Después de conseguir el *array*, se devuelven los valores almacenados tras indicar los canales que serán necesarios durante la ejecución del programa, aunque antes es necesario asegurar que los datos han sido almacenados de manera contigua.

5.5. Turnos de juego

5.5.1. Elección de ciega

La organización del código a la hora de elegir una ciega funciona de una manera muy similar a la mencionada antes. Una vez realizado el cambio de escena, el *TASBot* procede a hacer una captura de pantalla del programa.

Con esta imagen, el código hace una búsqueda para localizar el botón de selección de ciega, desplazando el ratón hasta el punto central del mismo, donde procede a pulsar el elemento. Este paso es igual en todas las escenas para marcar el paso de una a otra.

[1] Conjunto de píxeles que conforman una imagen
[2] Variable con un formato de listado que puede almacenar diversos valores

La única situación donde el código no selecciona de inmediato el botón es cuando la siguiente ciega se corresponde con un *boss*. En este caso, el código realizará una captura de pantalla de la zona donde se sitúa el identificador del *boss*, lo comparará con las imágenes almacenadas dentro del proyecto y extraerá el nombre concreto de la puja.

5.5.2. Jugar la mano

Dentro de la escena de juego, al comenzar la ronda el *TASBot* primero recorre las cartas. La información de cada carta (valor, palo, edición, encantamiento, sello) se almacena como atributos en un objeto 'card'.

Tras haber leído la información de las cartas, esta se almacena en un *array* de objetos '*card*' que representará la mano de cartas del jugador. Y, con esta mano, el *TASBot* analizará las opciones que puede realizar en ese turno y escogerá la mejor entre todas.

El *TASBot* seleccionará las cartas que quiere usar en su siguiente jugada, hasta un límite de cinco, y utilizará o descartará esa mano. Antes de realizar un descarte, comprobará si es posible conseguir una buena jugada en el futuro con las cartas que tiene en la mano. En el caso de que no sea posible, escogerá las cinco peores cartas para retirarlas. Por otro lado, si el *TASBot* decide jugar una mano, antes realizará varias operaciones para poder almacenar la cantidad de puntos que obtendrá con esa jugada. Si consigue superar el límite de puja, el código mandará una señal para cambiar a la escena de la tienda. Si no, se regresará a la escena de menú, desde donde se retomará el bucle de juego normal.

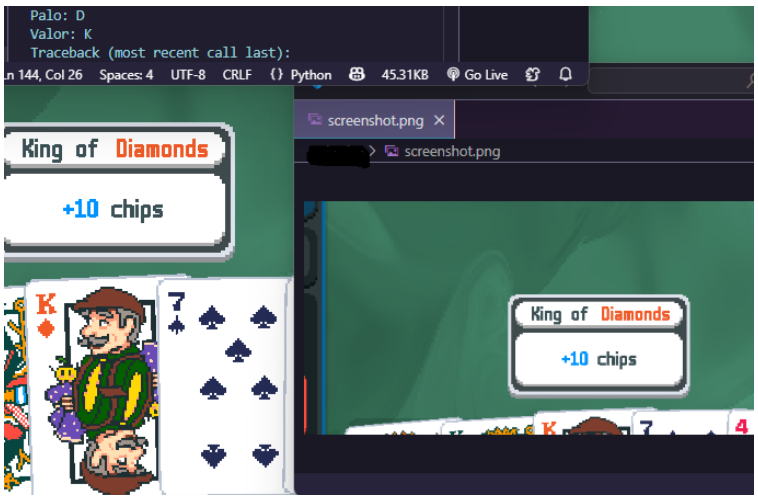

Figura 5.3: Diagrama que muestra el ejemplo de reconocimiento de una carta, estando en la parte de arriba como el programa recibe el Palo y el valor de la carta, en la parte de la derecha es la captura con la que lo procesa y a la izquierda la imagen del juego actual.

Capítulo 6. Arquitectura

6.1. Introducción a la Arquitectura

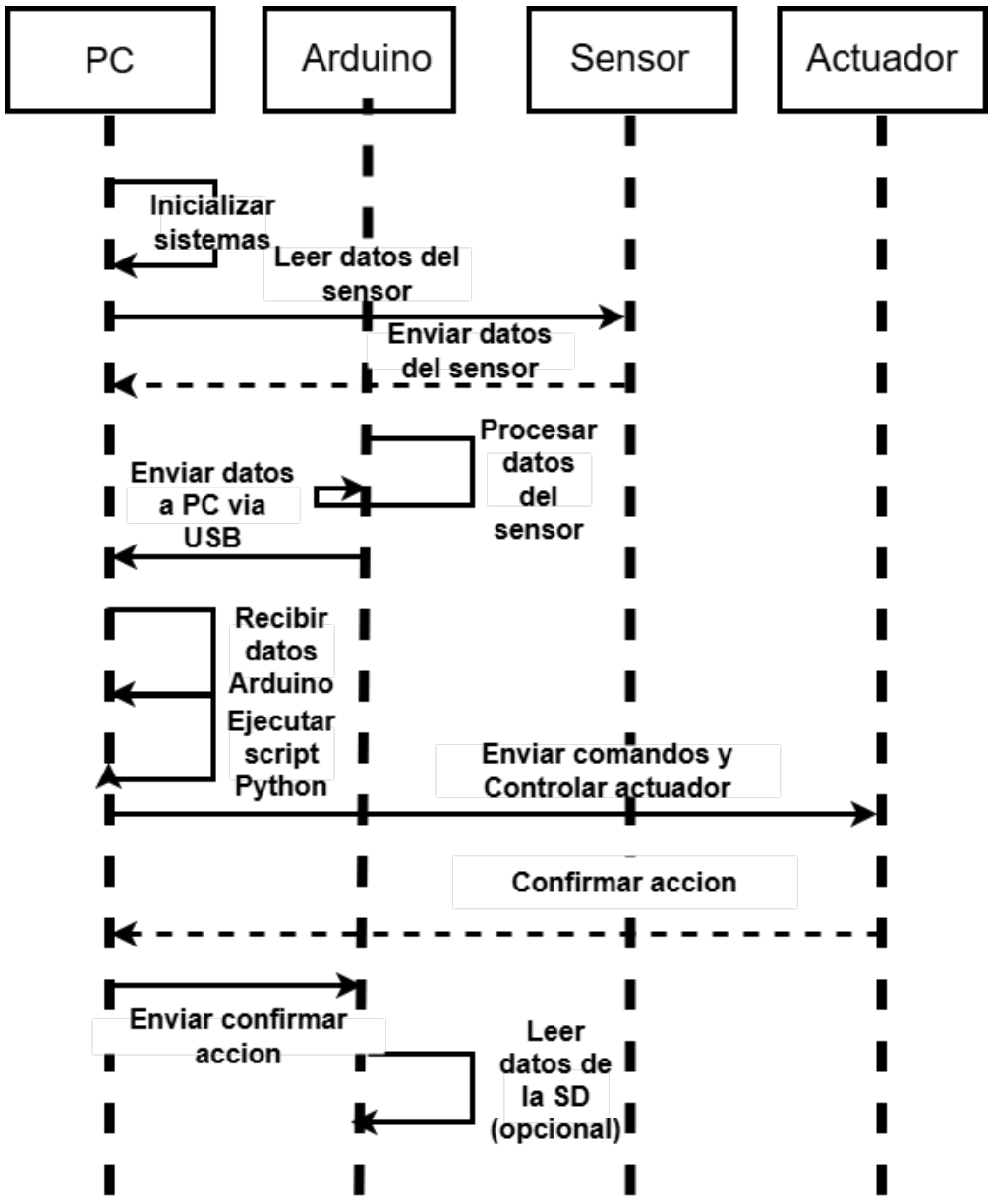

Figura 6.1: Diagrama general de la arquitectura con todos los pasos que se realizan, marcando las diferencias entre cada dispositivo

La arquitectura (mostrada en la Figura 6.1) está basada en una conexión mediante señales entre un *Arduino* (el cual se trata de un *Arduino UNO R3 WiFi ATmega328P+ESP8266*) y un ordenador con la ayuda del sensor para captar las señales que se manden de un dispositivo a otro y un actuador para que confirme las acciones al propio *Arduino*.

El diagrama 6.1, explicado paso por paso, comienza el proceso desde una placa de *Arduino* encargada de inicializar los sistemas. Una vez que han sido iniciados, el *Arduino* procede a leer los datos procedentes del sensor. La información procesada es transmitida al ordenador a través de un puerto USB y, a partir de dichos datos, comienza la ejecución de los *scripts* de *Python* involucrados. Después de ser ejecutados, el ordenador espera una señal de confirmación por parte del *Arduino*, quien la recibirá desde un 'actuador' encargado de controlar que las operaciones han sido realizadas con éxito.

6.2. Interconexión Arduino con PC

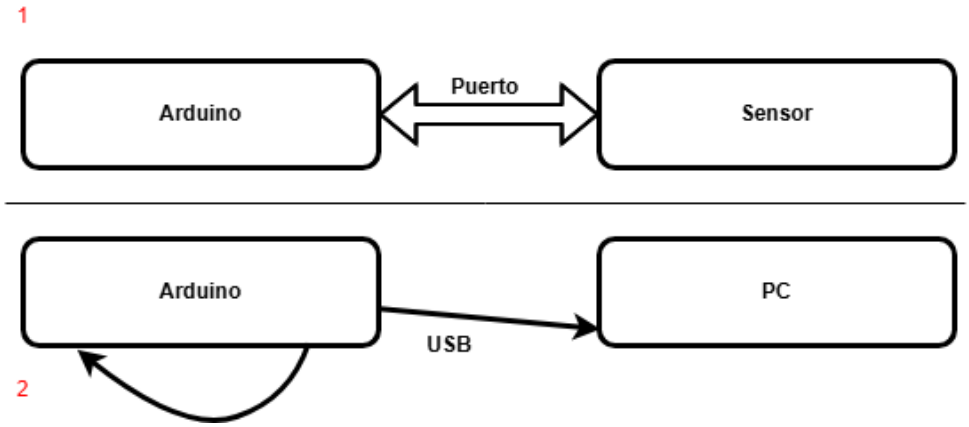

Figura 6.2: Diagrama general de los 3 pasos de inicio para iniciar la conexión Arduino-PC con ayuda de sensores que son quienes perciben el paso de información entre ambos extremos y los puertos que son el método de comunicación con ayuda del USB

Al principio, cuando el *Arduino* es encendido, intenta leer los datos del sensor, el cual trabaja como punto intermedio entre ambos extremos tratando de conectarse con el puerto del ordenador. El ordenador, una vez que recibe la señal de intento de conexión, manda datos procedentes del sensor para confirmar si hay una conexión. Una vez formalizada, el *Arduino* procesa estos datos para enviar una señal al ordenador mediante el USB (usando el *chip CH340G* cuya función es la comunicación USB-Serie) enviando una serie de datos para formalizar y señalar al ordenador de que empiece el proceso del propio *TASBot* y al juego por partes separadas.

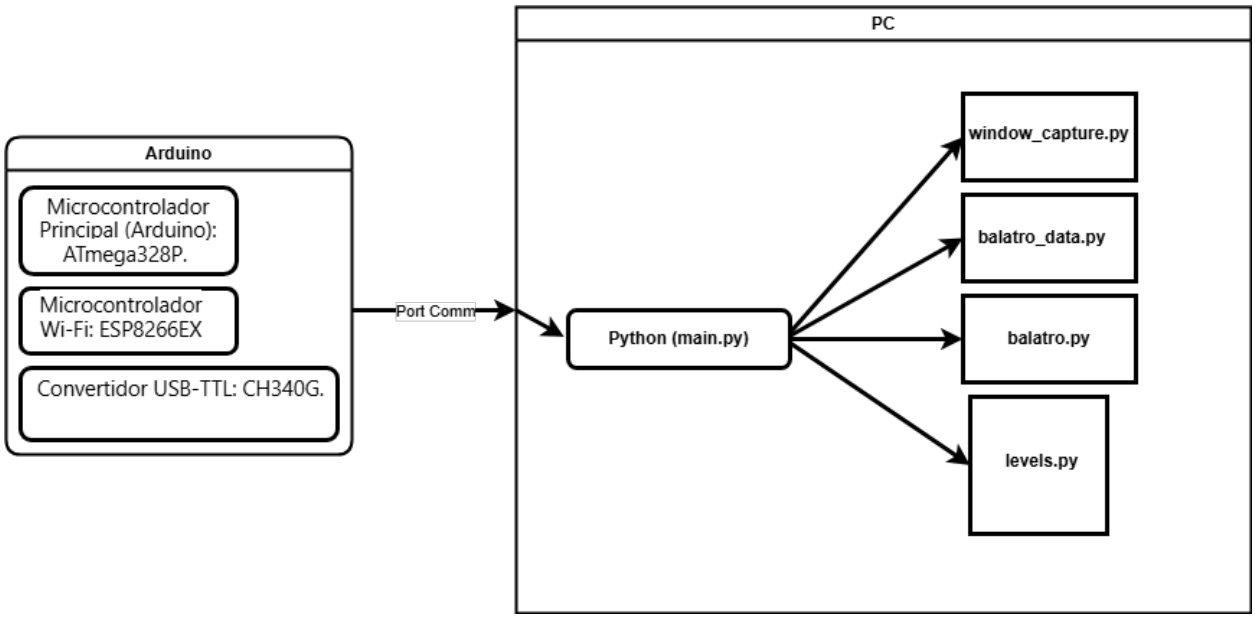

Figura 6.3: Diagrama general de la arquitectura con información de los diferentes módulos dentro de *Python* y su interconexión entre ellos

En este diagrama 6.3 se muestra cómo, una vez que se ha confirmado la conexión del *Arduino* con el propio ordenador, la comunicación se haría de manera directa con el propio código que funciona como administrador del juego. Por ello, el *TASBot* funciona como canal de comunicación con la placa de *Arduino*, recibiendo ayuda de un microcontrolador principal *ATmega328P* y un convertidor USB-TTL: CH340G.

Entre los módulos se encuentra *'window capture.py'*, encargado de gestionar las funciones de captura de imagen; *'balatro data.py'*, módulo que contiene todos los métodos necesarios para extraer la información necesaria de las capturas de pantalla para poder ejecutar los eventos de lógica del *TASBot*; *'balatro.py'*, donde se gestionan todas las funciones de lógica del juego, siendo el módulo que almacena la información de las cartas, los comodines o las rondas; y *'levels.py'*, encargado de administrar el transcurso de las escenas y el cambio entre ellas.

6.3. Captura y reconocimiento de Imagen para diferenciar escenas

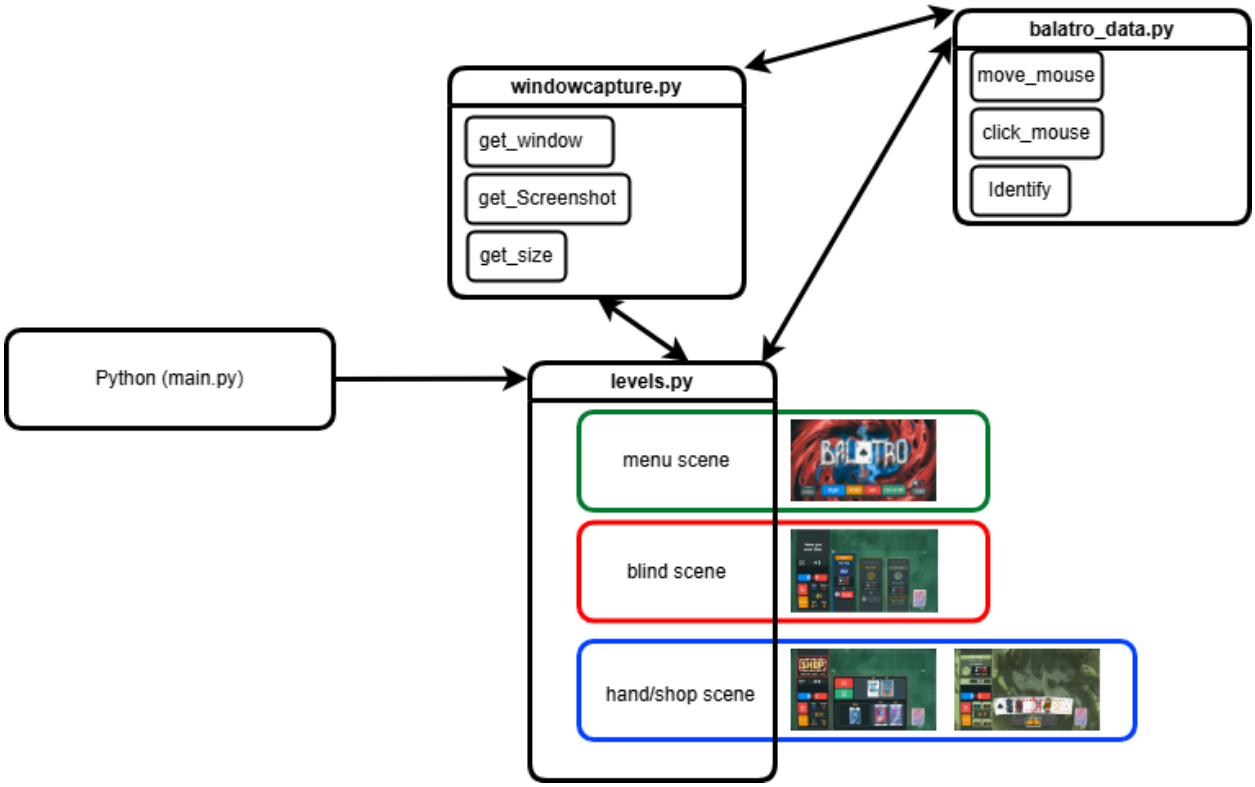

Figura 6.4: Diagrama general de la arquitectura centralizada en el reconocimiento de imágenes con todas sus conexiones

Como se observa en la Figura 6.4, el *script* principal *'main.py'* delega la gestión de las escenas del juego al módulo *'levels.py'*. Dentro de *'levels.py'* existen métodos específicos para cada una de las escenas del juego, que contienen las instrucciones que deben ser ejecutadas antes de que pueda ser realizado el cambio (por ejemplo, obtener información de una porción de la pantalla, interactuar con un elemento en específico de la pantalla o esperar unos segundos). Para poder interactuar con la ventana del juego (es decir, emplear eventos de captura de pantalla, movimiento del cursor o selección de elementos), *'levels.py'* utiliza funciones de *'windowcapture.py'* y, para la lógica específica de interpretación de datos del juego 'Balatro' se apoya en *'balatrodata.py'*, que contiene métodos destinados a obtener datos procedentes de las cartas, los comodines o los consumibles que luego podrán ser transmitidos a los eventos de gestión de lógica.

Una vez se inicia el ciclo de juego con *'menu scene'* (representado con el color verde dentro del diagrama 6.4) dentro de *'levels.py'* se pasaría a hacer

un intercambio de información entre las funciones de *'windowcapture.py'* y *'balatrodata.py'* para hacer las diferentes funciones de juego, como hacer clic sobre el botón. En el momento de terminar el bucle de *'menu scene'* pasaría al de *'blind scene'* (indicado con el color rojo en la Figura mencionada) y posteriormente al de *'hand scene'* y *'shop scene'* (ambas señaladas con el color azul en el diagrama).

Dentro de **'levels.py'** se pueden encontrar los siguientes métodos:

- ***'menu scene'***: Esta función inicia una partida, seleccionando uno de los mazos y escogiendo la dificultad, todo ello dependiendo de los parámetros que hayan sido establecidos en el código de manera previa. Tras haber indicado el tipo de partida que desea jugar, el programa envía un mensaje para ejecutar el método *'blind scene'*.

  Para seleccionar los parámetros de la partida, el método cuenta con una serie de instrucciones encargadas de realizar un reconocimiento de imágenes, utilizando unos archivos almacenados en el programa para ello. Una vez identificado uno de los elementos guardados en imagen dentro del juego, se llaman a métodos declarados en *'balatrodata.py'* que permiten desplazar el cursor y seleccionar dentro de la ventana del videojuego. Después pasaría a llamarse a *'blind scene'*.

- ***'blind scene'***: Se invoca esta función cuando se comienza una nueva partida o el *TASBot* abandona la escena de la tienda. En este método, con ayuda de un identificador de 'puja', se selecciona el nivel que debe ser jugado para continuar con la progresión del juego, empleando los métodos de *'balatrodata.py'* para ejecutar las acciones del cursor. Una vez escogido el nivel, el programa inicializa el método *'hand scene'*

- ***'hand scene'***: Este método contiene la lógica principal del juego, siendo el encargado de reconocer qué jugadas son las mejores dependiendo de las cartas que ha conseguido, los comodines de los que dispone y las condiciones que hayan sido establecidas con anterioridad en el *TASBot* por el jugador (Por ejemplo, que busque solo combinaciones de un mismo palo o que no utilice cartas concretas).

  Este método permanecerá en ejecución hasta que el *TASBot* logre superar la pantalla o pierda. En el caso de ganar, lanzará una instrucción para ejecutar la función *shop scene*; sino, mandará una

señal para volver a ejecutar el método *menu scene*, no sin antes cerrar la ventana con estadísticas de la partida.

- ***'shop scene'***: Si el *TASBot* ha logrado superar uno de los niveles, se llamará a la función *shop scene*. En este método existen varias instrucciones encargadas de la obtención de comodines o mejoras gracias a las funciones declaradas en *'balatrodata.py'* para el reconocimiento de imágenes, existiendo además varias condiciones para dar prioridad unos consumibles por encima de otros.

  Una vez terminado el proceso de compra, el *TASBot* seleccionará el botón para continuar con la partida, volviendo a llamar al método de *'blind scene'* para poder escoger un nuevo nivel.

En el fichero de **'windowcapture.py'**, por otro lado, están los métodos:

- ***'get window'***: Esta función recibe una variable que representa el nombre de la ventana que se quiere capturar. Una vez que ha sido localizada, los datos procedentes de la ventana son almacenados para su posterior uso en el programa, ya sea en otros módulos o en métodos dentro de *windowcapture.py*.
- ***'get size'***: Este método permite devolver los atributos de tamaño de la ventana que ha sido capturada con antelación, siendo estos valores el ancho y el alto.
- ***'get absolute coor'***: Teniendo como atributos de la función la propia ventana y un valor de posición, situado en un punto del eje X y un punto del eje Y, este método devuelve una serie de coordenadas absolutas, necesarias para establecer donde debe ser situado el cursor para seleccionar un elemento en pantalla.
- ***'get screenshot'***: Con esta función se lanza una ventana secundaria con el nombre de *Display*, encargada de mostrar *screenshot* de los elementos presentes en el videojuego que ha sido guardada mediante una copia del mapa de *bits*. Gracias a esta ventana, el *TASBot* puede mandar señales para ejecutar instrucciones de reconocimiento, que permitirán después seleccionar elementos en la ventana del videojuego.

Dentro del fichero **'balatrodata.py'** tenemos el grueso de los métodos encargados de identificar y almacenar la información de los elementos existentes en el juego, siendo los métodos:

- ***'click mouse'***: Esta función está encargada de desplazar el cursor hasta una posición 'X' e 'Y' establecida como variable durante su declaración. Una vez situado el cursor en las coordenadas correctas de la ventana del juego, se lanza la instrucción encargada de hacer *click*.

  Para poder mover el cursor de posición y seleccionar uno de los elementos en pantalla, el método necesita el módulo *'pyautogui'*.

- ***'identify hand'***: Este método recorre las cartas que han sido generadas del mazo, identificando el valor y el palo de cada una, además de si tienen una mejora, una edición o un sello establecido, guardando el identificador concreto de los datos mencionados si los tuvieran.

  Tras conseguir la información de una carta, esta será guardada dentro de un *array* que será proporcionado a los métodos encargados de gestionar la lógica de las partidas, volviéndose a llamar a este método una vez haya sido generada una nueva mano de cartas.

- ***'identify jokers'***: Esta función recorre todos los comodines situados en el área designada. De manera similar a lo que sucede en el método *'identify hand'*, en *'identify jokers'* se identifica cada comodín, guardando dentro de un *array* el nombre, el valor almacenado y su precio de venta, además de si posee una mejora o no, guardando el identificador en el caso de que los tengan.

  Debido a que cada comodín es único, una vez establecido su nombre, dependiendo de cual sea, su precio de venta, su rareza y su valor cambian, siendo el 'valor' una variable utilizada para guardar el dato acumulativo que puede poseer el comodín.

  Una vez que ha sido almacenada la información de todos los comodines dentro del *array*, este será proporcionado a los métodos encargados de gestionar la lógica del juego, llamándose de nuevo al método tan solo cuando vuelve a iniciarse un nivel.

- ***'identify consumable'***: Este método examina los consumibles presentes en la escena, identificando el nombre del objeto y el tipo de consumible. De la misma manera que sucede con las cartas en la mano y los comodines, la información obtenida de los consumibles se guarda en un *array* que será enviado a los métodos encargados de gestionar el árbol de decisiones del *TASBot*.

- ***'identify boss'***: Este método solo se utiliza cuando el siguiente nivel de la selección de ciega corresponde con un *'boss'*. Su función

es identificar el nombre de la penalización que tendrá el nivel, devolviendo el nombre para que la lógica pueda ejecutar las instrucciones asociadas con él.

- ***'identify voucher'***: Esta función solo puede ser llamada por *'shop scene'*, debido a que los billetes solo aparecen en la escena de tienda del videojuego. Las instrucciones del método permiten que pueda identificar el billete de la tienda, enviando el nombre obtenido para que la lógica del *TASBot* sea capaz de decidir su curso de acción.
- ***'select shop item'***: Este método solo puede ser utilizado por *'shop scene'* y está encargado de recorrer las zonas de la ventana donde pueden adquirirse comodines o consumibles. En el caso de poseer suficiente dinero, el programa ejecuta instrucciones para desplazar el cursor hasta los elementos en pantalla y seleccionarlos, actualizando la información almacenada de los billetes, los consumibles, las cartas y los comodines.

6.4. Interconexión con módulos de datos

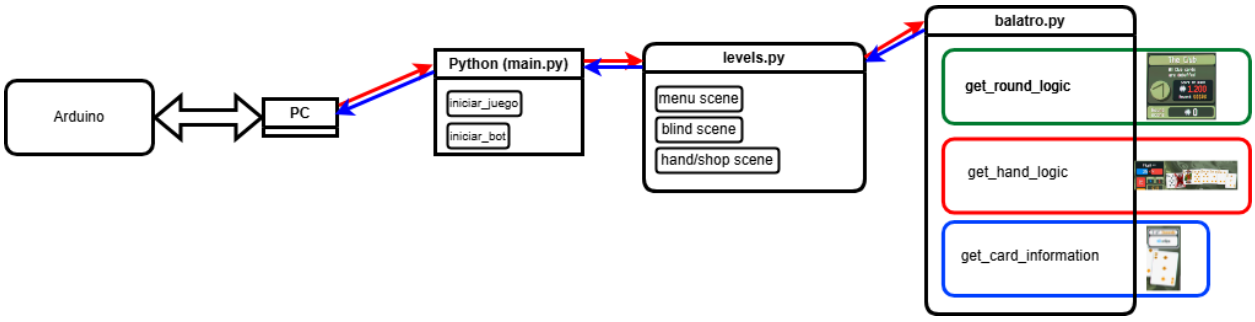

Figura 6.5: Diagrama que muestra el transcurso desde Arduino hasta el propio 'balatro.py' con las funciones que manejan las escenas

Una vez que los datos de la escena han sido procesados, los datos son enviados a un *script* llamado *'balatro.py'*, encargado de gestionar la mayoría de las instrucciones de lógica que posee el *TASBot*, entre las cuales se pueden hallar métodos destinados al tratamiento de los datos recibidos, la gestión de las cartas o la identificación de los valores de la ronda.

En el diagrama 6.5 se puede apreciar que el proceso comienza con el envío de señal desde una placa de *Arduino*. Un ordenador recibe susodicho mensaje a través de un puerto USB y, tras confirmar que la señal recibida es correcta, inicia dos subprocesos: 'iniciar juego', encargado de comenzar

a ejecutar el videojuego cuya ventana se desea capturar; e 'iniciar bot', encargado de comenzar la ejecución de los *scripts* que conformarán el bloque lógico del *TASBot*. Este último tendrá a su disposición un *script* llamado 'levels.py', que tendrá funciones destinadas a gestionar la lógica de cada

Por estos motivos, **'balatro.py'** es el bloque de código *python* más relevante a la hora de recibir los datos de la ronda o enviar información acerca de la mano disponible. Los otros módulos van a usar funciones de este *script* cuando necesiten procesar datos de las cartas o de la propia lógica de la ronda.

Dentro del archivo **'balatro.py'** se pueden encontrar los métodos:

- ***'get card information'***: Esta función analiza los datos obtenidos de la carta para extraer el valor, el palo, el número de *chips* y el multiplicador que alberga. Los *chips* dependen del valor de la carta mientras que el multiplicador en nulo, aunque dichos resultados pueden ser modificados dependiendo de la edición que posea la carta.

  Una vez que han sido obtenidos los datos de la carta, estos son devuelto con una variable para que puedan ser accedidos por otros métodos de *'balatro.py'*.

- ***'get hand type'***: Este método analiza las cartas de la mano para extraer todas las combinaciones posibles que pueden ser realizadas, calculando el número de *chips* y el valor del multiplicador que se pueden obtener en cada uno de los casos.

  Tras haber sido seleccionado el mejor resultado entre las opciones planteadas, se devuelve una variable que almacena las cartas que deben ser escogidas y el tipo de jugada que jugará el *TASBot*, para que otros métodos de la lógica puedan acceder a dicha información.

- ***'get hand logic'***: Esta función termina de calcular el resultado que obtendrá una determinada jugada, recibiendo los datos extraídos por el método *'get hand type'* para saber que cartas serán empleadas y que jugada ha sido utilizada. Para poder modificar el número de *chips* y el multiplicador, esta función recorrerá la lista de comodines y, una vez comprobados todos, devolverá el valor total de la jugada, el dinero conseguido, el nombre de la mano jugada y las cartas que ha tenido en la mano.

- ***'get round previous logic'***: Antes de que pueda ejecutarse la instrucción de *'get hand logic'*, el código manda una señal a esta función, encargada de gestionar los valores generales del inicio de la ronda, como puede ser el identificador del nivel, el número de descartes o la cantidad de cartas que puede haber en la mano.
- ***'get hand posterior logic'***: Tras jugar una mano, el código manda una señal a este método para modificar los comodines que son afectados después de cada jugada y actualizar el estado del mazo de descarte
- ***'get round posterior logic'***: Cuando el *TASBot* realiza una jugada que supera el valor de la puja establecida, el código llama este método para poder actualizar los comodines que son afectados después de cada nivel, obtener consumibles si existen cartas con el sello correcto en la mano y cambiar el identificador del siguiente nivel.
- ***'get discard logic'***: Este método está encargado de actualizar el mazo de cartas descartadas y modificar los comodines que son afectados después de que se haya realizado un descarte.
- ***'get discard cards'***: Este método tiene como función analizar las cartas que hay en la mano, con el objetivo de seleccionar las cartas que pueden ser descartadas. Para ello, sopesa las situaciones que pueden darse con la mano restante, dando prioridad a unas estrategias por encima de otras. Por ejemplo, el programa descartará un 'Rey' antes que un '2' si este último pertenece a un grupo de cuatro cartas de un mismo palo.
- ***'get consumibles logic'***: Esta función gestiona las modificaciones realizadas por los consumibles, pudiendo actualizar el valor de las manos, añadir mejoras a las cartas o generar un efecto que puede afectar a la mano.

Capítulo 7. Conclusiones y trabajo a futuro

7.1. Pruebas realizadas

Para poder comprobar que la conexión entre una placa de *Arduino* y un ordenador es correcta, se ha diseñado un programa que, tras detectar la presencia de una ventana especificada en el código del *script*, ejecuta una serie de instrucciones con el objetivo de superar una partida del videojuego del que se quiere realizar un *speedrun*. En este caso, el videojuego escogido para realizar las pruebas ha sido *Balatro* (LocalThunk, 2024).

El razonamiento detrás de dicha decisión está en el formato cíclico de las partidas, que siempre siguen la misma estructura. El jugador comienza con una puja que debe superar, obteniendo puntos cada vez que juega una mano de cartas. Si se consiguen los puntos necesarios, el videojuego carga la escena de la tienda, donde el jugador podrá incrementar sus probabilidades de victoria gracias a la obtención de comodines, cartas especiales o mejoras. Una vez realizadas las adquisiciones, el jugador vuelve a seleccionar una puja y a repetir el bucle de juego. Además, otro de los motivos que fomentaron realizar las pruebas con *Balatro* (LocalThunk, 2024) ha sido su componente de aleatoriedad, que provoca que cada partida sea única, y su gran variedad de opciones, que generan un número extenso de combinaciones y resultados.

El objetivo principal es la ejecución de un *TASBot* a través de una placa de *Arduino* que permita superar una serie de niveles del videojuego elegido, teniendo como meta secundaria conseguir que el programa logre finalizar una partida de manera exitosa.

Para ello, se debe conectar una placa de *Arduino* en el ordenador con un cable. El código de *Arduino* lanzará una instrucción para ejecutar el *script* de inicialización del juego y del propio *TASBot* que usará un algoritmo de detección de pantalla que, una vez haya ubicado la ventana de *Balatro* (LocalThunk, 2024), enviará un mensaje para lanzar los comandos procedentes del *TASBot*, programado con este juego en mente.

7.2. Objetivos realizados

A continuación, se indican los objetivos que han sido completados durante los meses que ha durado la realización del proyecto, todos ellos divididos según las diferentes fases del desarrollo en las cuales han sido completados:

- **Investigación del Medio**: Se realizó un trabajo de investigación acerca de los *speedruns*, su relación con la industria de los videojuegos y sus aplicaciones, estando todos estos puntos descritos en el Capítulo 2.
- **Investigación de Herramientas para Diseñar Speedruns**: Se buscó información acerca de librerías gráficas, con el objetivo de detectar e identificar los elementos que haya en pantalla, además de investigar varios dispositivos *Hardware* que facilitan a los jugadores superar el desafío impuesto o que permiten la ejecución automatizada de *speedruns*. Los resultados de la investigación están expuestos en el Capítulo 3.
- **Diseño de la Estructura e Implementación de Librerías**: Se diseñaron los ficheros encargados de realizar la conexión entre la placa de *Arduino* y el ordenador, además de implementarse los métodos de las librerías de detección gráfica.
- **Desarrollo de un TASBot**: Se programó la lógica del *TASBot* encargado de realizar *speedruns* del *Balatro*, además de conseguir enlazarlo con la ventana del juego para que pudiera seleccionar los elementos que fuese identificando de la pantalla sin emplear coordenadas explícitas.
- **Validación y Demostración**: Se realizaron varios vídeos durante las etapas del desarrollo del proyecto, con el objetivo de mostrar el establecimiento de conexión entre los elementos y el paso de escenas en el juego, conforme va superándolas el *TASBot*.

7.3. Trabajo a Futuro

Durante los meses que ha durado el desarrollo del proyecto, uno de los objetivos secundarios con mayor relevancia era conseguir que el programa fuese capaz de superar el desafío impuesto por el desarrollador del *Balatro*, sin importar el mazo seleccionado, los comodines o la dificultad establecida. Aunque el *TASBot* diseñado funciona, el código podría ser mejorado con el objetivo de completar una partida en su totalidad.

Otra de las mejoras que han sido contempladas para realizar a futuro es el diseño de un *TASBot* que permita realizar un *speedrun* de un videojuego en 3D, donde sea necesario tener en cuenta la profundidad de los escenarios a la hora de programar el árbol de decisiones del programa.

Por último, se baraja la posibilidad de desarrollar un programa basado en este que permita realizar la conexión no solo entre una placa de *Arduino* y un ordenador, sino entre un *Arduino* y una consola, como puede ser una *Nintendo Switch*, una *PlayStation* o una *X-Box*.

Capítulo 8. Conclusions and future applications of the project

8.1. Test performed

To test that the connection between the Arduino plate and the computer is correct, the program has been designed that, after detecting the presence of a window specified in the code of the script, executes a series of instructions with the objective of overcoming or completing the game that has been selected to do the speedrun. In this case, the videogame we have chosen to test it was *Balatro* (LocalThunk,2024)

The reasoning behind the decision is in the cyclical format of the game's rounds. The player starts with a blind that must be reached, obtaining points from the different hands it has played. If the player reaches the points necessary, the videogame loads the scene for the shop, where the player can increment their chances to win the game thanks to the jokers, special cards or upgrades that can be bought in that scene. Then the player chooses again the blind to reach and it repeats the game loop. Also, another reason that helped to do the test with *Balatro* (LocalThunk,2024) was the component of randomness, which provokes each play of the game to be unique and the great variety of options that generates a big number of combinations and results.

At first, the main goal is the execution of the *TASBot* through the *Arduino* plate that allows one to win a series of levels, leaving as a secondary objective to achieve that the program reaches the end of the game in a successful way.

To do that, it must be connected to the *Arduino* plate with the computer using a cable. The *Arduino* code will launch an instruction to execute the *script* to initialize the game and the *TASBot* that will use an algorithm of screen detection that, once it locates the window of *Balatro* (LocalThunk, 2024) it will send a message to launch the needed commands of the *TASBot*, programmed with this game in mind.

8.2. Achieved Objectives

Next, the objectives that have been completed during the months of work for the project will be indicated, each one divided following the

different phases of development in which they have been completed.

- **Investigation of the Medium**: A work of research was done about speedruns, the relation with the videogame industry and their applications, all of those points are described in the Chapter 2.
- **Investigation of the Tools to Design Speedruns**: Information about the different graphic libraries were searched with the objectives to detect and identify the elements that were on the screen, but also investigate different Hardware devices that makes it easier for the player to pass the challenge imposed or that allows an automated execution of the speedruns. The results of this investigation are exposed in Chapter 3.
- **Design of the Structure and Implementation of the Libraries**: We designed files in charge of realizing the conection between the *Arduino* plate and the computer but also the implementation of methods of the libraries of graphic detection.
- **Development of the TASBot**: The logic of the *TASBot* in charge of realize the speedruns of *Balatro* was designed, in addition to getting it linked with window of the game so it could select the elements that it was identifying of the screen without using the explicit coordinates.
- **Validation and Demostration**: Various videos were made throughout the different stages of the development of the project with the objective of proving the stablishment of connection between the elements and the passage through scenes in the game, as the *TASBot* completes each scene.

8.3. Future Work

During the months of development of the project, one of the most relevant secondary objectives was to achieve that the program was able to overcome the challenge imposed by the developer of *Balatro*, without taking into account the deck selected, the 'jokers' or the difficulty established.

Even though the *TASBot* works, the code can be subject to changes or upgrades with the objective of completing a game in its totality.

Another of the upgrades that were considered for implementation in the future was the design of a *TASBot* to allow the speedrun to be in a 3D

videogame, where it's necessary to take into account the depth of the scene when programming the decision tree.

Finally, we are considering the possibility of developing a program based on this one, that allows the connection between *Arduino* and other consoles like a *Nintendo Switch*, a *PlayStation* or a *X-Box*.

Bibliografía

[1] I. Mussa and J. Messias, "Code breakers e bunny hoppers: transgressão inventiva nas camadas intra e suprajogo," *Sessões do Imaginário*, vol. 22, no. 38, pp. 14–27, 2017.

[2] F. Barnabé, "The transformative power of speedrun: Deconstruction and recodification of pokémon games' communicative structures," *978-3-948791-20-9*, 2021.

[3] S. T.-H. Sher and N. M. Su, "Speedrunning for charity: How donations gather around a live streamed couch," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–26, 2019.

[4] F. Messaoudi, G. Simon, and A. Ksentini, "Dissecting games engines: The case of unity3d," in *2015 international workshop on network and systems support for games (NetGames)*. IEEE, 2015, pp. 1–6.

[5] F. Fummi, M. Loghi, S. Martini, M. Monguzzi, G. Perbellini, and M. Poncino, "Virtual hardware prototyping through timed hardware-software co-simulation," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 798–803.

[6] U. Rivera-Ortega, "Interactive projectile motion stem simulation and game, based on scratch (s4a) and arduino," *Physics Education*, vol. 56, no. 6, p. 065029, 2021.

[7] K. Lohse, N. Shirzad, A. Verster, N. Hodges, and H. M. Van der Loos, "Video games and rehabilitation: using design principles to enhance engagement in physical therapy," *Journal of neurologic physical therapy*, vol. 37, no. 4, pp. 166–175, 2013.

[8] J. Halton, "Virtual rehabilitation with video games: A new frontier for occupational therapy," *Occupational therapy now*, vol. 9, no. 6, pp. 12–14, 2008.

[9] R. Rosman, M. Z. A. Hadi, and N. A. Bakar, "Interactive design and development of real arm movements for application in rehabilitation," in *IOP Conference Series: Materials Science and Engineering*, vol. 341, no. 1. IOP Publishing, 2018, p. 012007.

[10] M. Mediawan, M. Yusro, and J. Bintoro, "Automatic watering system in plant house-using arduino," in *IOP Conference Series: Materials*

Science and Engineering, vol. 434, no. 1. IOP Publishing, 2018, p. 012220.

[11] S. Sharma and S. Harish, "Xy-drawing robot using arduino," *College of Engineering, Pune-Satara Road, Pune*, vol. 411043, no. 6, p. 2, 2019.

[12] K. M. Udofia, "Arduino microcontroller-based intelligent car parking system," *Journal of multidisciplinary Engineering Science and Technology*, vol. 7, no. 3, pp. 1–8, 2020.

[13] J.-L. Tseng and C.-W. Chu, "Interaction design in virtual reality game using arduino sensors," in *Simulation and gaming*. IntechOpen, 2017.

[14] F. Gerardi and G. Grisetti, "Custom wireless joystick development with arduino and linux integration," 2024.

[15] R. Scully-Blaker, "Re-curating the accident: Speedrunning as community and practice," Ph.D. dissertation, Concordia University, 2016.

[16] M. Groß[1], D. Zühlke, and B. Naujoks, "Check for," in *Applications of Evolutionary Computation: 25th European Conference, EvoApplications 2022, Held as Part of EvoStar 2022, Madrid, Spain, April 20–22, 2022, Proceedings*, vol. 13224. Springer Nature, 2022, p. 471.

[17] A. Meades, "Why we glitch: process, meaning and pleasure in the discovery, documentation, sharing and use of videogame exploits," *Well Played Journal*, vol. 2, no. 2, pp. 79–98, 2013.

[18] A. Dongare, R. Kharde, A. D. Kachare *et al.*, "Introduction to artificial neural network," *International Journal of Engineering and Innovative Technology (IJEIT)*, vol. 2, no. 1, pp. 189–194, 2012.

[19] A. Francés Lillo, "Artificial intelligence for videogames with deep learning," 2019.

[20] G. Lami, "Speedrunning and path integrals," *arXiv preprint arXiv:2403.13008*, 2024.

[21] T. Fort, "Controlling randomness: Using procedural generation to influence player uncertainty in video games," 2015.