Herramienta adaptativa para la creación de tests automáticos de interfaces de videojuegos
Adaptative tool for automatic test creation of video game interfaces

Trabajo de Fin de Grado
Curso 2022–2023

Autor
Cristian Rene Castillo de León
Iago Quintas Diz

Director
Guillermo Jiménez Díaz

Colaborador
Alessandro Lentini

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Herramienta adaptativa para la creación de tests automáticos de interfaces de videojuegos
Adaptative tool for automatic test creation of video game interfaces

Trabajo de Fin de Grado en Desarrollo de Videojuegos
Departamento de Software e Inteligencia Artificial

Autor
Cristian Rene Castillo de León
Iago Quintas Diz

Director
Guillermo Jiménez Díaz

Colaborador
Alessandro Lentini

Convocatoria: *Septiembre 2023*
Calificación:

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

15 de Septiembre de 2023

Dedicatoria

A mi familia, por apoyarme y ayudarme en todo lo posible, no lo habría conseguido sin vosotros. A mi madre, por darme todo lo que necesitaba para cumplir mis sueños. Y a Raquel, que has sido mi faro durante tantos años, por escucharme y conseguir sacarme una sonrisa todos los días.

Cris

A Aaron, por haber estado siempre ahí a lo largo de estos años y por ser un gran amigo. A mis padres, por haber confiado en mí y darlo todo para que hoy este aqui. A mi hermano, por haber sido siempre mi mejor amigo e inspirarme a ser mejor. Y a mi hermana, que siempre consigue sacarme una sonrisa.

Iago

Agradecimientos

Gracias a todo el equipo por habernos dado todo lo que necesitábamos. Gracias a Alessandro y Angela por sus ideas y seguimiento. A Claudio por motivarnos a trabajar todos los días tratando siempre de aportarnos ideas y preocupándose por nosotros. A Blanca y a Ricardo por ser nuestros referentes y estar siempre disponibles cuando les hemos necesitado. A Fran, por ayudarnos a adaptarnos al entorno y estar ahí para apoyarnos. A la gente que aunque no trabajase con nosotros directamente, se interesó por el trabajo que haciamos. Y finalmente a Guille, por haberse preocupado tanto por el proyecto y habernos dado la oportunidad de acceder a estas prácticas. A todo el mundo que confió en el proyecto o dedicó parte de su tiempo en seguir nuestro progreso, gracias.

Resumen

Herramienta adaptativa para la creación de tests automáticos de interfaces de videojuegos

El *testing* es una parte fundamental durante el desarrollo de un programa o aplicación de software. Este proceso consiste en verificar y validar las distintas funcionalidades de un programa y de esta forma conseguir un producto de calidad. Sin dicha parte la mayoría de las aplicaciones serían inutilizables debido a la cantidad de errores que experimentaría un usuario.

Sin embargo, el *testing* requiere de mucho tiempo y dinero para lograr unos resultados significativos, llevando a las empresas de videojuegos a contar con un gran número de *testers* que comprueban manualmente el funcionamiento de sus productos. En el caso de interfaces de videojuegos, las nuevas tecnologías y herramientas de automatización pueden ser útiles para reducir este trabajo manual y dedicarlo a pruebas únicamente realizables por humanos.

Usando aprendizaje automático y preprocesado de imágenes, se ha hecho una herramienta capaz de etiquetar de forma general elementos de una interfaz de videojuegos aplicable a múltiples títulos para la realización de pruebas de navegación automáticas.

Palabras clave

automatización, preprocesado, clasificación, detección, interfaz, aprendizaje, videojuegos, testing, bordes, adaptable

Abstract

Adaptative tool for automatic test creation of video game interfaces

Testing is a fundamental part of software program or application development. This process involves verifying and validating the various functionalities of a program in order to produce a quality product. Without this step, most applications would be unusable due to the number of errors that a user would experience.

However, testing requires a lot of time and money to achieve significant results, leading video game companies to rely on a large number of testers who manually check the functioning of their products. In the case of video game interfaces, new technologies and automation tools can be useful in reducing this manual work and dedicating it to tests that humans can only perform.

Using machine learning and image preprocessing, a tool has been developed, being capable of detecting and generally labelling elements of a video game interface that can be applied to multiple titles for the purpose of automated navigation testing.

Keywords

automation, preprocessing, classification, detection, interface, learning, videogames, testing, edges, adaptability

Contents

1. Introduction 1

1.1. Motivation 1

1.2. Objectives 2

1.3. Work plan 3

1.4. Document outline 4

2. State of the art 5

2.1. Graphic user interfaces on video games 5

2.2. GUI testing 8

2.2.1. Functional testing 9

2.2.2. Visual testing 11

2.3. Computer vision techniques 12

2.3.1. Optical character recognition 12

2.3.2. Tesseract 14

2.3.3. OpenCV 14

2.4. Image classification 14

2.4.1. Convolutional Neural Networks for image classification 15

2.5. Conclusion 16

3. Design of a tool for creation of automatic tests of video game interfaces 17

3.1. Cogito pipeline 17

3.2. Image processing 18

3.2.1. Adaptative contrast and brightness 19

3.2.2. Adaptative noise and blur filtering 20

3.2.3. Edge detection . . . 24
3.2.4. Contour detection . . . 27
3.2.5. Text detection . . . 28
3.2.6. Icon detection . . . 29
3.3. Element classification . . . 31
3.3.1. Text classification . . . 31
3.3.2. Icon classification . . . 31
3.3.3. Meaningful element groups . . . 32
3.4. Choose the element to interact . . . 35
3.5. Conclusion . . . 35

4. Development **37**
4.1. First research and prototypes . . . 37
4.1.1. Tesseract text detection . . . 37
4.1.2. GameDriver functional testing . . . 38
4.1.3. Sikuli visual testing . . . 38
4.1.4. Amazon Web Services (AWS) . . . 39
4.1.5. Prototype of a machine learning model . . . 41
4.2. Iterative development . . . 41
4.2.1. First Iteration - Text detection . . . 41
4.2.2. Second Iteration - Image processing . . . 42
4.2.3. Third Iteration - Element classification . . . 44
4.2.4. Fourth Iteration - Checkers and Highlighted Button . . . 47
4.2.5. Fifth Iteration - Lambda . . . 48
4.3. Architecture . . . 48
4.3.1. Image Processing module . . . 49
4.3.2. Element Detection module . . . 50
4.3.3. Element Classification module . . . 50
4.3.4. Meaningful Element Identification module . . . 51
4.3.5. Configuration module . . . 51
4.4. Conclusion . . . 52

5. Evaluation **55**
5.1. Quality performance . . . 55

5.1.1. Methodology 55
5.1.2. Results and conclusions 56
5.2. Time performance 58
5.2.1. Methodology 58
5.2.2. Results and conclusions 58
5.3. Conclusions 59

6. Conclusions and Future Work **65**
6.1. Future work 66

Bibliography **69**

A. Appendix A - Contributions **71**
A.1. Cristian Rene Castillo de Leon 71
A.2. Iago Quintas Diz 73

List of figures

1.1. Technical debt increases over production time 1

2.1. Example of figure-ground principle (Todorovic, 2008). 6

2.2. Example of proximity principle (Todorovic, 2008). 6

2.3. Example of similarity principle (Todorovic, 2008). 7

2.4. Example of closure principle (Todorovic, 2008). 7

2.5. Example of continuity principle (Todorovic, 2008). 7

2.6. Comparison between a standard HUD and a diegetic HUD 8

2.7. Example of test programmed on Selenium IDE 10

2.8. Example of test programmed on Sikuli IDE 12

2.9. Simplification done by OCRs from text to bit images (Smith, 1987). 13

3.1. Cogito pipeline 18

3.2. F1® 22 multiplayer menu example 19

3.3. Image Processing pipeline 19

3.4. F1® 22 menu before and after binary threshold 20

3.5. F1® 22 adapted contrast and brightness result 20

3.6. Sobel intensity graphic (Bradski and Kaehler, 2008) 22

3.7. Second derivative of an edge (Bradski and Kaehler, 2008) 22

3.8. Gaussian filter application on contrasted image, the entire image is blurred in order to obtain the most prominent shapes in the image. 23

3.9. Median filter application on contrasted image, a minimal blur to reduce the noise in the image is applied. This modification of the image results in smoothed parts, such as the face of the figure. 24

3.10. Bilateral filter application on contrasted image. This filter helps smoothing out the noise in specific parts while preserving edges such as the background and buttons in this figure. 25

3.11. *One-dimensional edge profiles.* (Jain et al., 1995) 25

3.12. HSV Multi-Channel Canny result 26

3.13. Comparison of Canny applied to a Non-blurred and Gaussian filter F1® 21 image. Contours and edges from inside each detected element are moderately reduced on the right after the filter is applied. 27

3.14. Comparison of Canny applied to a Non-blurred and Median filter F1® 20 image. The noise is slightly reduced for the tool to correctly detect the elements on the screen. 27

3.15. Comparison of Canny applied to a Non-blurred and Bilateral filter FIFA 23 image. Most of the noise from the background is removed. 28

3.16. Filtered contours of a F1® 2020 image 28

3.17. Bounding boxes of a F1® 2020 image 29

3.18. Text detected on a STAR WARS Jedi: Survivor™ menu 29

3.19. Icons detected by machine learning model on a FIFA 23 settings menu . . . 30

3.20. Text detection and icon detection performed on a FIFA 23 menu. Left being the original image and right being a visual representation of each kind of element detected. 30

3.21. Exit and Settings label dictionary examples 31

3.22. Classified texts on a STAR WARS Jedi: Survivor™ menu. The green boxes outline the detected texts and display the label classified on top of them. . . 32

3.23. Classified icons on a FIFA 23 settings menu. The green boxes outline the detected icons and display the classified label and confidence on top of them. 32

3.24. Horizontal checker for menu bar detection on a F1® 2021 menu 34

3.25. Example result of all the final information gathered by the tool. 35

4.1. Tesseract text detection prototype simple example 37

4.2. Tesseract text detection prototype FIFA 23 menu example 38

4.3. Sikuli test example 39

4.4. Example of Rekognition text detection. 40

4.5. SageMaker first button detection model 41

4.6. Comparison between AWS text detection services on Madden. 42

4.7. Missing contours in a FIFA 23 settings menu 43

4.8. Artificial contours for missing contours detected 43

4.9. Example of generic icon's dataset and its use on FIFA 23 44

4.10. Example of screenshots of different menus from FIFA 23 46
4.11. Combined dictionary result example. This example shows the type of element, the text it contains with its position and dimensions as well as the position and dimensions of its contour, and finally its labels. 47
4.12. Cogito UML architecture diagram 49
4.13. *Labels.json* example containing two labels. 52
4.14. *Appsettings.json* example. It contains the game to be tested, whether lambda will be used for processing or locally and whether debug images will be generated. 53

5.1. Confusion matrix for different steps of the tool through F1® 2022 screenshots 57
5.2. Table of times of each step of the tool applied to FIFA 2023 screenshots (in seconds) 61
5.3. Table of times of each step of the tool applied to F1® 2022 screenshots (in seconds) 62
5.4. Graph of times of each step of the tool applied to FIFA 2023 screenshots (in seconds) 63
5.5. Graph of times of each step of the tool applied to F1® 2022 screenshots (in seconds) 63
5.6. Boxplot of times of each step of the tool applied to FIFA 2023 screenshots (in seconds) 63
5.7. Boxplot of times of each step of the tool applied to F1® 2022 screenshots (in seconds) 64

List of tables

Chapter 1

Introduction

"Cogito ergo sum"
— René Descartes

1.1. Motivation

Manual testing of video games can be a time-consuming and costly process, as it requires testers to play games repeatedly to identify and report any bugs or issues that may arise. The large scale of these projects make it a very complex task to detect all these issues. For that reason, the cost of the technical debt should be minimized (Keith, 2015) in order to avoid these increased solving costs that accumulate between all development stages.

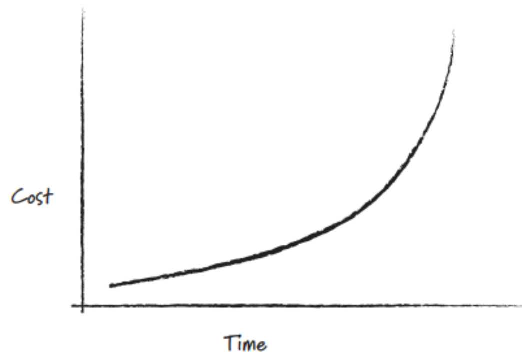


Figure 1.1: Technical debt increases over production time

As Figure 1.1 shows, the more advanced the development stages are, the more cost it will be required to solve issues with the code. Fixing these issues on earlier stages can take very little time once they are found, However, if they are not fixed, the cost to solve these bugs grows exponentially.

Advancements in technology have led to the development of even more sophisticated automated testing tools, such as AI-powered testing, which can help further reduce the costs and time associated with manual testing. Different AI approaches such as natural processing language or neural networks are interesting technologies that could be really useful if integrated with an actual testing workflow, making testing more efficient.

While automated testing can be a more cost-effective and efficient option, it is important to note that these tests still require human oversight. However, the goal is to trust humans to do the final assertions of tests so they can still be reliable, while the computer does the rest of the tedious work, the navigation and all the back end implementation on how to understand what is happening on a specific moment.

Overall, the use of automation in video game testing is becoming increasingly popular, as it can help reduce costs and improve the efficiency of the testing process. However, it is important to have a balance between automated and manual testing, as both have their advantages and limitations.

One specific area where testing is needed is interfaces. It can highly benefit of automatic testing due to the nature of the tests: doing an specific set of actions to arrive to certain menus, check button functionalities or even to check if a menu has not critical bugs that make the video game unplayable.

Usually, graphic user interfaces, or GUIs, are designed with usability principles on mind. For this reason, if a relationship could be established between these principles and the testing of what is seen on the screen, it is possible that a better understanding of what is happening on the tests would be obtained.

1.2. Objectives

The main goal of this investigation is contributing to the development of Cogito, a tool capable of performing automatic testing of interfaces on video games by introducing human like actions. An example could be the "Go to the Settings Menu" action, Cogito will be able to identify the different user interface elements of the video game and find the specific elements to reach that menu. The tool will identify the different elements using a combination of text, contours, edges and icon detection and then it will classify those elements in different labels using text and image classification.

This main goal is divided into three sub-goals to be completed in order:

1. Research about already existing tools and possible approaches. Some investigation will be made about alternative testing software: What functionalities do they provide and how are those implemented.
2. Design and development of the tool.
3. Evaluating the effectiveness and performance of the tool.

The work will be mainly focused on the part related to analyze the information on the screen. Given a screenshot of the current state of a video game, the tool should detect all important detected elements, being those:

- Classifying each button of the interface into a meaningful label about its role.
- Detect selected buttons, if any.
- Return the grouped elements like: grids, menu bars and lists.

One of the most important points on the development of the tool will be its adaptability, to be able to work in different types of games and interfaces with the lowest possible maintenance. Since most of the existing tools require a lot of maintenance or are built inside the game itself, this solution will work outside of any engine or game.

There are some limitations which define what the project will focus on:

- A fully focused machine learning solution is not necessary. Some steps for the project can be achieved through the use of image processing algorithms, due to the complexity of a fully generic machine learning element detector.
- The interfaces where the tool will be tested will be the subsequent ones to the main menu interface. Those interfaces that are a part of the game will not be tested, as well as diegetic interfaces that are a part of the game's world.

The project will be built as a tool to improve automatic interface testing for Electronic Arts and will allow testers to focus on performing less tedious and repetitive tests.

1.3. Work plan

This project has been developed based on the SCRUM methodology. SCRUM is one of the most widely used methodologies in the software development industry, a project management method (Sutherland, 2015). Following this methodology, it has been established that the work plan will be divided into two-week periods called sprints, in each of the sprints the team will meet to assign each of the members a number of tasks, each task will have an estimated score that will indicate the complexity of the task and a priority to indicate which tasks should be done before others. At the end of each sprint, the resolved tasks will be closed and those still in progress or to be started will be assigned to the next sprint, and new ones will be created. Due to the complexity and continuous research of the project, there will not be many tasks in the beginning and more will be created over time.

Before starting to work with the team for the creation of the tool, the project will focus on the research of the different existing tools for the creation of tests and will investigate Amazon's machine learning tools.

The development of the tool will start with a period of a few weeks of research on SageMaker and Rekognition services by creating different image recognition models and observing the advantages and disadvantages of these models. Once the models have been trained and tested with refined datasets, a prototype of how the tool will work will be developed, without yet focusing on navigation and focusing only on the recognition of interface elements. Based on the results of these image recognition models, the team will study the different alternatives or combinations to be implemented for the final tool.

The team has already determined part of the technologies that will be used during the development of the tool, mainly Amazon Web Services (AWS) such as SageMaker or Rekognition for the machine learning parts of the project. Python will be used as the development language, Microsoft Visual Studio Code as the environment and the OpenCV library for image processing. The reason for using Python as a programming language is because it allows fast iteration between prototypes. In addition, when closed it will allow

the code to run on a server using AWS Lambda technology, thus allowing to continue to call AWS services with a single credentials per server.

Throughout the development of the project, code inspections will be carried out to improve the tool, and as the project progresses and achieves results, ways to improve efficiency will be explored.

1.4. Document outline

This document is structured as follows:

1. Chapter 1 Introduction, describes the motivations and objectives of this project.
2. Chapter 2 State of the Art, explains the current testing technologies related to this project, different computer vision techniques, machine learning algorithms and optical character recognition solutions.
3. Chapter 3 Tool Design, describes the functionality and capabilities of the tool. It focuses mainly on the workflow and processes performed for the correct operation of the tool.
4. Chapter 4 Tool Development, describes the entire project development process along with the problems and solutions that occurred during this process.
5. Chapter 5 Evaluation, explains the evaluated capabilities in real test cases to determine the correct functionality of the tool, including performance evaluations.
6. Chapter 6 Conclusions and Future work, analyzes the results of this project and what improvements can be made or will be made in the future.

Chapter 2

State of the art

In order to create an useful tool for testing of interfaces, an investigation has been done prior to working on the design of the tool. This chapter explains what are graphic user interfaces on video games and what logical rules they often follow. It also describes the current situation of testing: the differences between functional and visual testing.

This chapter also revises some of the software currently used related to these approaches. Describes computer vision techniques and their relation with optical character recognition and machine learning. Related to machine learning, deep learning models that can classify data, Convolutional Neural Networks for image classification and other machine learning models for text classification are explained.

2.1. Graphic user interfaces on video games

Before focusing on the different approaches to test user interfaces, they should be defined, and in specific, for video games. A user interface (UI) is defined by Cambridge (Cam, 2023) as:

> *The way in which the information on a computer, phone, etc. and instructions on how to use it are arranged on the screen and shown to the user.*

Graphic user interfaces (GUI) can be displayed as an infinite combination of colors and shapes. They do not necessarily need to follow many rules, which causes that defining what is the standard for GUIs a complicated task.

Although UIs can be pretty different between themselves, they are ideally made with a main objective in mind: they have to be clear for the user. If an interface is not easily understandable, it is very probable the user gets frustrated, and stops using its related software. On the contrary, if the user can understand what is on the screen only taking a glance at it, the user will have a more satisfying experience. Due to this reason, good GUIs apply Gestalt principles of design (Gomez and Olfman, 2012).

Gestalt principles are rules of the organization of perceptual scenes. They are widely used to archive visual and comprehensive interfaces. The rules they are based on are the following ones:

- Figure-ground articulation. Humans perceive images instinctively as shapes. If everything displayed was reduced to its most simplified colors, there should still be recognizable shapes on it. That is what figure-ground principle is about. Well implemented GUIs should have a clear separation between the GUI elements and the background.

Figure 2.1: Example of figure-ground principle (Todorovic, 2008).

As Figure 2.1 shows, different colors are used to make a contrast between the green shape and the background.

- Proximity. When GUI elements are close to each other, forming groups, humans understand they are related somehow. This happens because there is an illusion of a superior group that is made of the elements that build it.

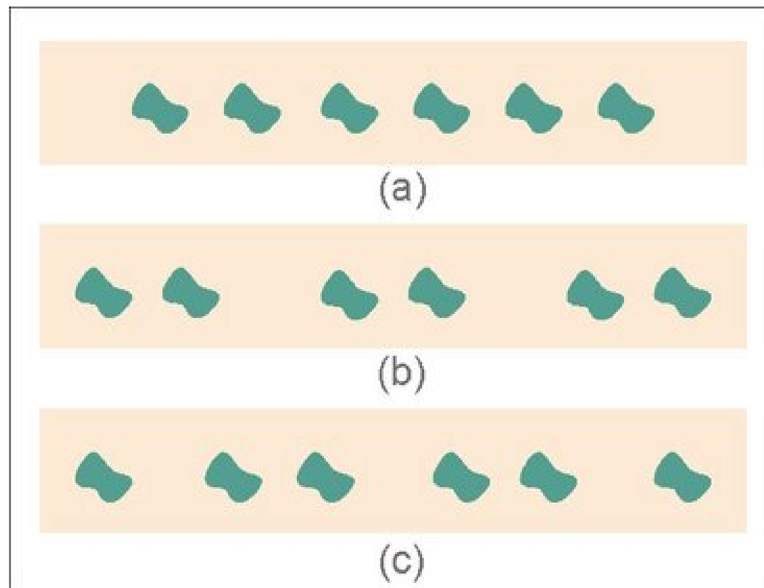


Figure 2.2: Example of proximity principle (Todorovic, 2008).

As Figure 2.2 shows, (a) is seen as a whole group, however (b) is seen as three groups 12/34/56, and lastly (c) is seen as 4 groups 1/23/4/6.

- Common fate. According to Gestalt laws, if elements move together with the same direction, they will be perceived as a group.

- Similarity. Elements that have characteristics in common are perceived as a group. Anything visual that can be recognized as a distinguishing feature. Some examples of it would be similar shape, color, size, orientation, transparency and many more.

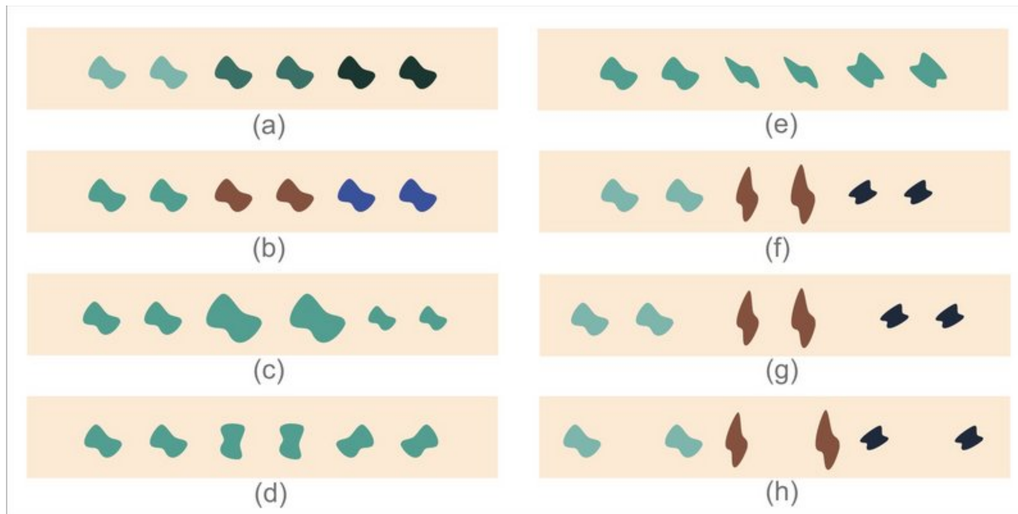


Figure 2.3: Example of similarity principle (Todorovic, 2008).

As Figure 2.3 shows, (a) has similarity in lightness, (b) in color, (c) in size, (d) in orientation, (e) in shape. (f) combines multiple similarities, (g) combines this law with the proximity one, getting a stronger effect, and (h) weakens its relation doing the opposite.

- Closure. When elements have a resemblance to a closed group, even if they are not entirely closed, they will be perceived as if that was the case.

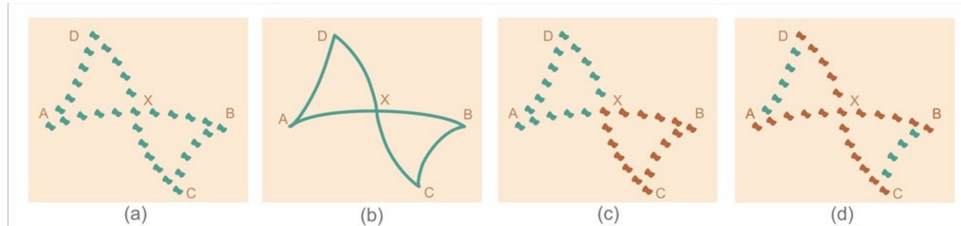


Figure 2.4: Example of closure principle (Todorovic, 2008).

As Figure 2.4 shows, (a) is the not closed group, (b) is how that group is perceived, and (c) and (d) are examples of this principle being overpowered by the similarity principle, weakening its effect.

- Continuity. When elements are aligned with each other they will be perceived as a group. It is similar to the closure law, but applied specifically to lines.

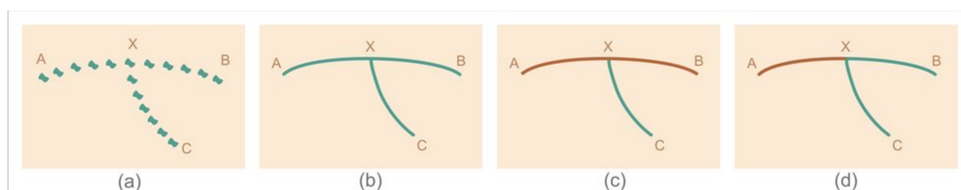


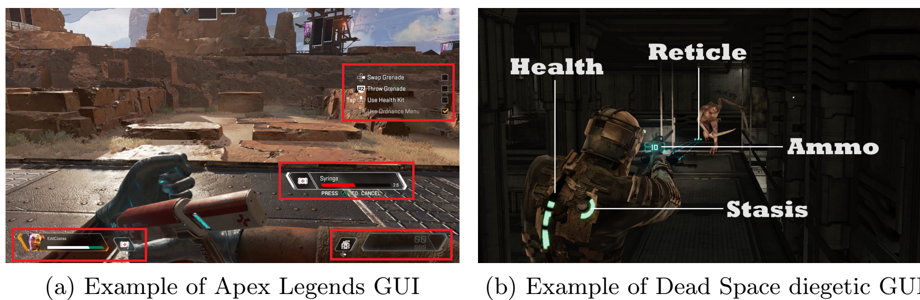
Figure 2.5: Example of continuity principle (Todorovic, 2008).

As Figure 2.5 shows, (a) is the original image, (b) is how it is perceived as continuity law says, (c) is how groups are perceived on image (a) due to their alignment, and

(d) is a way to break this principle effect by coloring the branches differently, using similarity law.

As it is obvious, video games GUIs also follow the Gestalt principles. Having the ability to understand what is going on a game that requires a fast response time is something necessary. Games like Apex Legends (Ape, Electronic Arts. Apex Legends. 2019) provide a quickly understandable interface that allows to read to current state of a game. Despite of that, as a last case, video games have another different point related to other software's GUIs. They heavily depend on aesthetics to make the game appealing for the player, as well as showing internal consistency. The use of stylized fonts or tilted texts is relatively common on game design.

When talking about GUIs it is important to highlight that this term refers mainly to two different things: HUDs (heads-up display) and menus. While HUDs are information that hovers in front of the gameplay, menus are composed of intractable buttons with the purpose of guiding the player through different actions in the game, such as playing, changing settings, communicating with friends, etc.



(a) Example of Apex Legends GUI (b) Example of Dead Space diegetic GUI

Figure 2.6: Comparison between a standard HUD and a diegetic HUD

In addition to that, some games integrate the interface in a diegetic way. Usually interfaces are displayed in an way that is external to the world of the game. On Figure 2.6 both cases are compared. On (a), red squares indicate parts of a HUD that are hovering in front of the gameplay, adding useful information. In contrast to these, diegetic interfaces are blended with the actual world instead of being displayed over the game, as it can be seen on (b). These last ones can be more complex to interpret.

2.2. GUI testing

Software testing is the process of checking that a software does what it has been designed to do. This does not mean testing should only check if software does its function, but also check for unexpected behaviour (Baresi and Pezzè, 2006). In addition, testing is not a phase, but an iterative process. After testing is done, the errors or bugs found should be labeled and solved, and once that happens, same tests should be done again to confirm the code is working correctly. It is a cycle that should be done throughout the whole development to guarantee the software works as close as intended. However, to achieve a software free of bugs can be unrealistic, even more on large and complex projects such as video games, where the time and resources needed to test features working together grows exponentially.

GUI testing happens when software testing is focused only on graphic user interfaces. This would include, but would not be limited to: checking if interactive elements do its function, information is displayed correctly respecting the intended format, the whole menus are reachable, and of course, if critical bugs do not happen and crash the software.

Testing can be categorized in many ways, but one common way to do so is separating them by what software part they focus on. With this in mind, two categories can be established: functional testing and visual testing.

2.2.1. Functional testing

Functional testing is a type of testing that verifies the functionality of an application by ensuring that the software is performing the tasks it was designed to do and meets the user requirements.

In functional testing, each function is tested by giving an expected value, determining the output, and asserting the actual output was correct. Functional testing is also called black-box testing because it focuses on application specification rather than actual code. The tester has to test only the program rather than the system.

However, this approach has some limitations:

- It only checks for unexpected behaviour on the functional side, as its name implies. If the application is displaying its visual elements in a wrong way, there is no way to check it.
- It is totally dependant on having internal access to the code. For this reason, it can not be tested on software which code is private. That is the situation for testers, who play to a specific build of the video game, without access to the code.

This way of testing is widely used. Many libraries exist in order to integrate functional testing into different environments. It is very common applied to websites, but there are also examples of it integrated on video game engines.

2.2.1.1. Selenium

Selenium (Huggins, 2004) is a tool used for web browser functional testing automation. This tool allows developers to write scripts for web navigation, click elements on the page, fill in forms, etc. Selenium supports the automation of all the major browsers in the market through the use of WebDriver, an API and protocol that defines a language-neutral interface for controlling the behaviour of web browsers. Each browser is backed by a specific WebDriver implementation, called a driver. The driver is the component responsible for delegating down to the browser and acts as a bridge to communicate between the automation script and the web browser.

Selenium includes three groups of instructions:

- Input actions, related to navigation and interaction with elements of the web page. Accessing to the HTML code, Selenium is able to perform actions on specific elements searching for its identifier or its properties.

- Waits, needed for the nature of the platform the tests are being run on. After doing an action, a delay should be expected until the website loads all the required elements. For this reason, there are two main ways to wait:
  - Active waits. Waiting for a specific time of seconds. Not very reliable due to short waits can both pass or fail a test, depending on the connection of the user.
  - Wait for an element. More efficient on successful tests, since once the element is found, the test can continue running.
- Asserts. In order to check if a test has passed successfully, there are multiple instructions to verify the test is correct. Some examples are if an element exists, if it is visible or if the title of the website contains specific words.

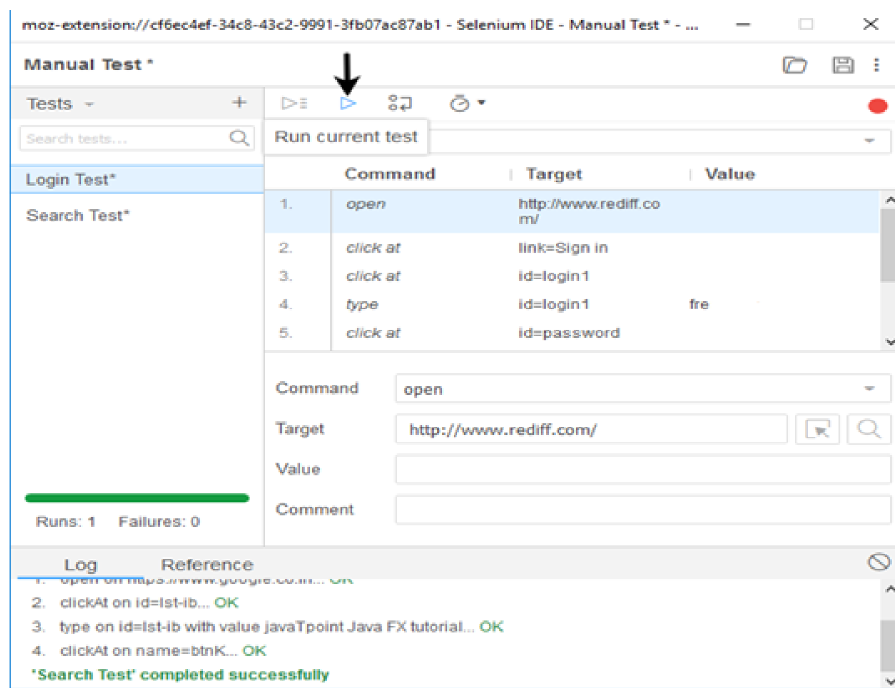


Figure 2.7: Example of test programmed on Selenium IDE

As Figure 2.7 shows, Selenium has an integrated development environment (IDE) that allows to easily create functional tests. There is a list of commands like the ones that have been described before, which are performed in order.

2.2.1.2. GameDriver API

GameDriver (Gutierrez et al., 2018) is a functional testing automation video game API designed to automate repetitive tasks a player must do in a video game. It was developed specifically for the Unity engine and allowed effective communication with the elements of the scenes. The API enables the tester to control and interrogate the game, asking about its properties such as the score, positions of elements, current scene, and more. GameDriver API enables the tester to create a script that waits for a specific time or an object, required for the test, to appear. The script can take control of the keyboard and mouse with specific commands and allows the capture of images and video. This API

also provides methods for moving objects or characters to a specific location and detecting elements on the screen by name and colour.

2.2.1.3. Appium

Appium (Cuellar, 2012) is an open-source tool that allows the development of automatic functional tests in both Mobile Web, iOS and Android applications. A single API that works on both platforms. Recently Appium has also been able to test Windows applications.

Appium allows the development of tests on Web Applications, as well as Native Applications on mobile devices. The main difference between Appium is the ability to perform automatic tests on both Android and iOS.

Appium uses an HTTP server written in Node.js that creates and manages multiple WebDriver sessions on Android and iOS devices, WebDriver is an automation framework that works with different APIs for sending commands to and interacting with an application or web.

The typical operation of Appium would be, a client or tester sends a request to the Appium server, the server will receive the request and handle it by sending it to the device or emulator that will execute the request or test cases requested by the client and send the results to the Appium server that will send it to the client for observation. It works similarly to Selenium, an HTTP server that listens to and sends your requests.

2.2.2. Visual testing

Visual testing, also known as visual regression testing, is a type of automatic testing that verifies the visual appearance of a software application by comparing screenshots of the application before and after changes are made. Visual testing ensures that the application's user interface remains consistent and visually appealing after changes are made. The main advantage of visual testing over functional testing is that it can detect visual defects that may be missed by functional tests. For example, functional testing may not detect a change in font size or colour that could affect the overall user experience. Visual testing can also help ensure brand consistency and compliance with design standards.

It has one great advantage compared to functional testing: it only relies on the visual output of an application, so it can be generic. One example of this is Sikuli, which uses computer vision techniques integrated with its tests.

2.2.2.1. Sikuli

Sikuli (Chang and Yeh, 2009) is an open source tool used to perform automatic visual testing. It is an interpreter for multiple scripting languages as Python, used to define tests cases. It also supports an API to use specific visual testing actions, for instance, searching a template image inside another screenshot. To implement all these actions, the computer vision library OpenCV is used.

As it is shown on Figure 2.8, tests are programmed on python, with the addition of an API that allows passing screenshots as parameters. There is an integration inside of the

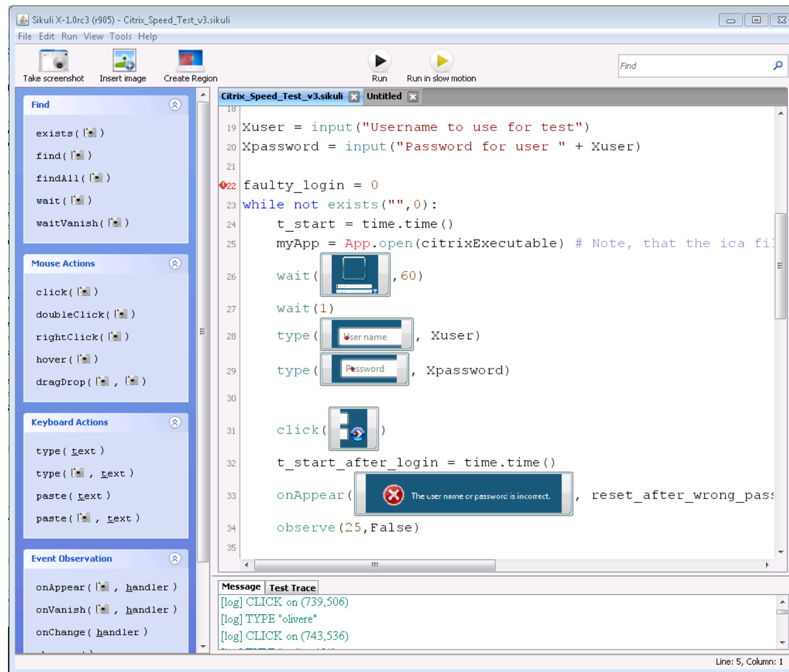


Figure 2.8: Example of test programmed on Sikuli IDE

tool that allows to capture the region of interest of the screen to speed up programming the tests.

Just like all the previously studied tools do in functional testing, it has wait instructions to give time to the program to change between states and load all the resources needed before next instruction. This kind of instructions are expanded with the image recognition API, allowing to also wait for an specific image to appear.

2.3. Computer vision techniques

Computer vision techniques involve the use of computers to view elements of an image in a similar way of how an actual human would see them. That way, computers can understand images, allowing to automatize tasks that previously only humans could do.

There are multiple examples of computer vision algorithms, such as edge detection, feature extraction, template matching or histogram comparison. Some of these will be explained along this chapter and the following ones (see subsection 3.2.3 for more details).

2.3.1. Optical character recognition

Optical character recognition, or OCR, is the process of converting texts from any kind of image data to encoded text that can be recognized by machines. It is one of the most logical steps to take related to visual testing. It is used in a variety of situations, such as its integration with automation tools.

One of its most common uses is being integrated in automatic workflow processes. AI-powered tools often rely on OCR either to obtain datasets by processing documents related

to one subject, or for obtaining input cases for other AI models to use them.

Written text detection, either by a machine or a human, depend on a lot of factors. Font size, color, contrast with the background, alignment or orientation can affect the whole detection. The existence of different fonts is a very troublesome factor, not to mention how text is read on different languages.

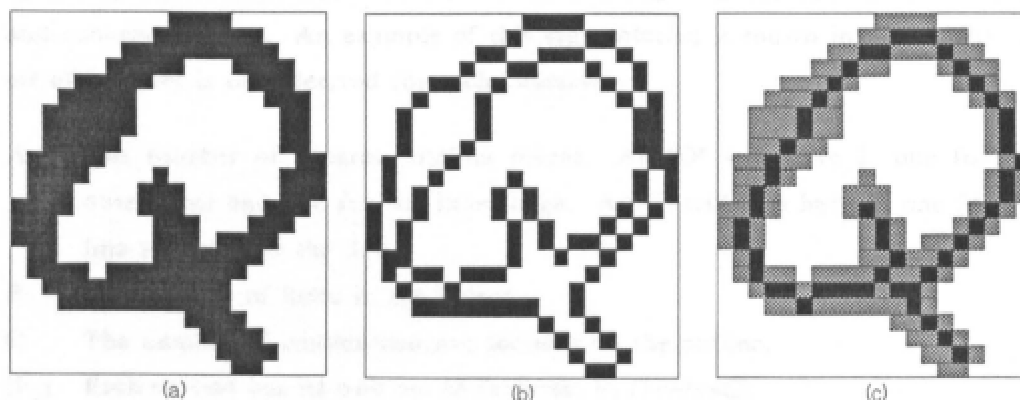


Figure 2.9: Simplification done by OCRs from text to bit images (Smith, 1987).

As Figure 2.9 shows, each letter is detected (a), then an skeleton of the image is obtained (b) and finally the skeleton is reduced to a minimum shape and stored as a bitmap.

The story of OCRs has been in constant development. When this technology was first investigated, only computer vision techniques were used. The simplification seen on Figure 2.9 was used later in an algorithm called **feature extraction**. It consists obtaining the most interesting points of an image in this case, in order to reduce the amount of data with the models are trained on. This algorithm selects the points, labeling them according to the following characteristics per section. Center point, concave or convex, distance to the central point or length are some of these characteristics.

In the past, OCR technology only depended on computer vision techniques, which had limitations. Even if feature extraction is a powerful tool, adaptability is not one of its strengths. In cases like handwritten text, where big enough variations can occur between different cases, it can lead to false negatives. Also, the more features detected per letter, the more challenging it will be computationally to analyze (Wang et al., 2021).

Nowadays every state-of-the-art OCR is based on machine learning. It makes the whole process easier, as well as saves computational time and efforts. However, computer vision is not excluded from the process. In order to obtain meaningful data from each letter, feature extraction is still used, but now is also done by the model. This is what is known as deep learning. Next, a model is in charge of assigning encoded text letter instead of comparison as it was previously done.

Related to computer vision, there are two open source libraries that are often used: Tesseract and OpenCV. Tesseract is an OCR library while OpenCV includes other computer vision techniques, standing out for its wide range of image processing functionalities.

2.3.2. Tesseract

Tesseract (Smith and Hewlett-Packard, 1984) is an open source engine OCR which can be used on different libraries, depending on the language. It is currently trained with more than 100 languages and most of the common fonts, and allows the use of neural nets for line detection on its last version, Tesseract 4. It also gives support to the previous version, which recognizes character patterns.

For the legacy engine, it requires .traineddata files, which include the result of the training for each language. In addition to the 100 languages it is trained on, Tesseract can be trained on other languages as well generating another .traineddata as has been done with the original supported languages.

Tesseract text works in unicode (UTF-8). It supports PNG, JPEG and TIFF files as input and multiple outputs such as plain text or PDFs.

2.3.3. OpenCV

Open Source Computer Vision Library, or OpenCV (Bradski and Kaehler, 2008), is a powerful open source computer vision and machine learning library that implements time efficient operations with images. It acts as a common infrastructure for machine learning and computer vision algorithms, including both classic algorithms and state-of-the-art ones.

OpenCV has been programmed using C++ and has bindings with Java, Python and MATLAB, among other languages. This allows to use efficient in time algorithms while keeping the fast iteration workflow Python allows.

OpenCV counts with its own deep neural network module (dnn) that allows to prepare data in order to use them for training machine learning models. Besides that, it also allows to perform image conversion to simplify data, operating with images in both RGB or HSV color schemes, as well as grayscales. This is particularly useful when it comes feed less information to models to obtain better performing solutions. However, this simplification can be also helpful with the classic computer vision algorithms in some cases, as it happens with edge detection algorithms.

2.4. Image classification

In order to automate visual testing of video game interfaces, it is necessary for the tool to somehow understand the meaning of the elements it will detect on the screen. One of the solutions most widely used to classify data at present is deep learning.

Deep learning is a field within machine learning. Often deep learning is referenced for images and feature extraction (defined on Figure 2.3.1). Additionally, in deep learning, the models automatically perform feature extraction of the characteristics with which better results would be obtained (MathWorks, 2017).

Deep learning is used on neural networks as it will be explained in detail on the next subsection.

2.4.1. Convolutional Neural Networks for image classification

Neural networks (NN) are structures based on how an actual brain works like. They are composed of interconnected neurons, the most basic unit of data, and those are grouped in layers, which are communicated with weights that modify the information shared between neurons. Each neuron is connected to all the neurons from the previous and next layer. Most often, the application for them is using them as classifiers, where first layer contains the input, and last layer is the output. Between these 2 layers there are hidden layers, which function is to find the correlations and patters between the input and the output, modifying the weights of the layers throughout the process.

The process of training a neural network consists on iterate throughout them using functions called forward propagation algorithms. These go from the input layers to the output ones, measuring the error obtained on the end and adjusting the weights of the neural network in order to get close to the desired result. In addition to that, there are some extensions to these process, such as back propagation, which allows to check the error in the output layer and send it back to the previous layers so the adjustment is more precise.

Convolutional neural networks (CNN) differ on typical neural networks on the way neurons communicate with each other. Input neurons are not all assigned to every hidden layer's neuron. Instead, each input neuron is assigned to a group of neurons from the hidden layer with weights that never change, since each hidden layer's neuron is only used for a specific feature. This way mimics how an actual brain works, and this process has been improved on updated versions like RNCC, Fast RNCC and Faster RNCC.

Their efficiency has been tested and work better with large volumes of data (Victor Ikechukwu et al., 2021). Specially deep convolutional neural networks, which are neural networks with multiple hidden layers. As it has been said on the previous section, using a pretrained neural network facilitates training, allowing training to be done with some initial weights and features, even if the labels for the images are different to the ones used originally to train the NN. This opens the possibility of training over neural networks that have already been trained on large datasets of generic images, and tune them into classiffiers with more specific labels. Currently, there are two popular ones widely used for this purpose:

- Residual neural network (ResNet-50) is a CNN already trained with more than one million images from the ImageNet dataset, and counts with 50 hidden layers. It appeared as a solution for the vanishing gradient problem, which made the optimization of neural networks harder as error increased. It uses a technique called skip connections, which allows to omit communications that would endanger the accuracy of the neural network.

- VGG-19 (Visual Geometry Group) is another CNN, trained with 19 hidden layers. It is based on a variation of the same dataset used by ResNet-50. Counts with 1000 labels and was trained over 150000 images. Its structure allows it to perform a better feature extraction, while using state-of-the-art technologies in the field of CNN.

2.5. Conclusion

This chapter has discussed the different ways of testing based on what they aim for, functional and visual testing. Also, a research has been made about the different open source libraries and machine learning models that can be useful for the development of the project.

Both these testing approaches have their own strengths, which will be discussed later in the design chapter. Although the software studied performs automatic tests following a guide script, they are not adaptative to changes in the application under test. Even when functional testing methodology is proven to be useful, they are not compatible with all testing workflows due to its previously mentioned limitations. However, it is important to look into detail about its capabilities given that they might be useful for future work.

Since good video game GUIs follow Gestalt principles, they constitute the environment that will appear more commonly on the majority of existing games. With this information a standard point can be guaranteed in order to perform tests on interfaces, knowing which elements they have in common.

Having studied the relevant technologies in this chapter, it is time to integrate them into the tool design.

Chapter 3

Design of a tool for creation of automatic tests of video game interfaces

The standard way to perform menu navigation testing is based on testing a video game's interface manually. Other methods include functional automation testing tools (subsection 2.2.1) or manually programmed scripts. All these solutions require a lot of work and maintenance, being tedious as they will have to be changed to test them in each new game. The use of automation in video game testing has become quite popular and menu navigation is no exception, the design of this tool takes a leap forward focusing also on adaptability, allowing automatic navigation in multiple titles using one unique solution without the need for tedious maintenance and extensive scripting.

This chapter will discuss the design of our tool for creation of automatic tests of video game interfaces, Cogito. Particularly, it will be focused on automatic menu navigation. This tool processes images to detect the different elements in a user interface using a combination of image processing, optical character recognition and machine learning solutions. The detected elements are then classified into different labels and based on the results Cogito finds the specific elements to reach the desired destination. Cogito will use a scripting language to enable high-level testing and will work in different types of video games.

The design of the tool was an ongoing process of research that changed significantly throughout the project. The final functionality and features of this tool will be explained during this chapter.

3.1. Cogito pipeline

Cogito is an adaptative tool for the automatic navigation of menus of multiple different titles. This tool uses a combination of different image recognition algorithms and machine learning models to detect and classify the elements on a video game screenshot and, based on the results, moves or clicks on the item that is most likely to help it reach the desired destination.

The Cogito pipeline works as follows:

1. Write a testing script indicating actions such as "Go to Settings".
2. Navigation, run the script on the game:
   a) Take a screenshot of the current state of the game.
   b) Identify and classify elements on the screen:
      1) Image processing
         - Adaptative contrast and brightness.
         - Adaptative noise and blur filtering.
         - Edge detection.
         - Contour detection.
         - Text detection.
         - Icon detection.
      2) Element classification:
         - Text classification.
         - Icon classification.
         - Meaningful element groups.
           - Element grouping detection
           - Highlighted button detector
           - Finding navegable menus
   c) Choose the element to interact with based on the previous steps.
   d) Repeat until reaching destination.

The main focus of this thesis has been the step (b): Identify and classify elements on the screen. Most of the work has been done on the identification and classification of data due to the large scope of the project. Outside of the main focus, step (c) has been briefly approached in a certain way which will be explained later on. Figure 3.1 represents the whole process.

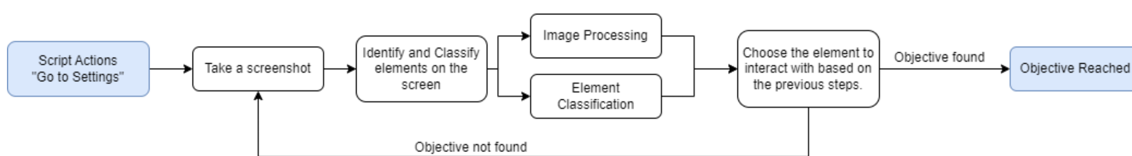


Figure 3.1: Cogito pipeline

The following sections describe in detail the entire Cogito process and features.

3.2. Image processing

The Image Processing section is one of the most important parts of the tool, as it helps to find the sizes and positions of the different interface elements.

It was concluded that in order to achieve an adaptative tool, it was necessary to find a way to focus only on the important elements of a menu, leaving behind everything unnecessary such as dynamic elements or backgrounds. As seen in Figure 3.2, most of the

interface elements have an unnecessary drawing or picture that does nothing to indicate the meaning of the element, but it is still required to get its whole size and position to achieve a proper navigation and menu structure, that is why image processing is used.

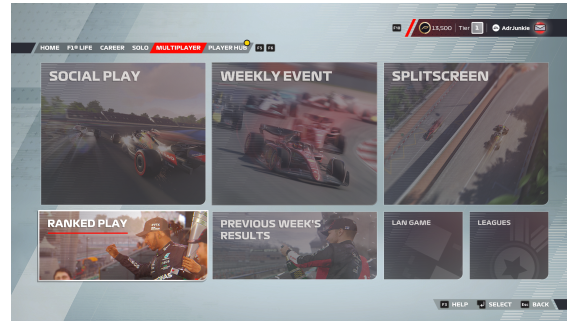


Figure 3.2: F1® 22 multiplayer menu example

This process consists mainly in the detection of edges and contours in a menu image, but for the tool to be adaptable to multiple types of games, an adaptative process of different filters must be performed in order to achieve the best results. Figure 3.3 represents the whole Image Processing pipeline.

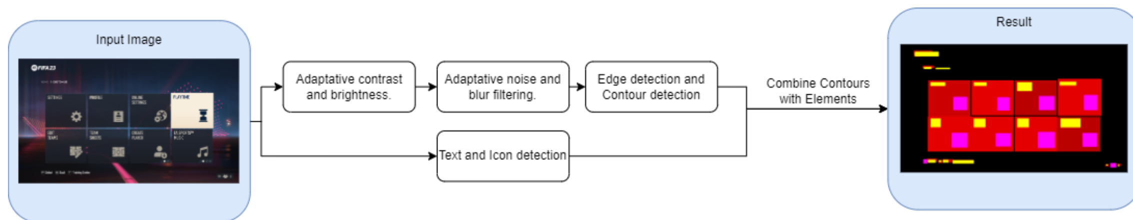


Figure 3.3: Image Processing pipeline

The following subsections describe in depth the Image Processing pipeline.

3.2.1. Adaptative contrast and brightness

Tonal contrast is the difference between bright areas and dark areas in an image. To achieve a solution adaptable to any type of interface, the best possible contrast and brightness settings must be achieved. The implemented proposal is based on the value of the tonal contrast calculation of the image, and thus be able to adjust the overall brightness and contrast of the image depending on that value.

This tonal contrast value is calculated and then used in a combination of operations:

1. The input image in RGBA is converted into gray-scale to apply a fixed-level threshold to get a binary image result. This fixed-level threshold converts any pixel that exceeds the threshold to white and those not exceeded to black.
2. Based on the binary result, the number of black and white pixels and thus the average tonal contrast of the image is obtained, as can be seen in Figure 3.4.
3. This average tonal contrast can then be used to obtain a brightness value to change the original image's brightness; the resulting image can be obtained by calculating

(a) Before binary threshold (b) After binary threshold

Figure 3.4: F1® 22 menu before and after binary threshold

the sum of the image weights by a given contrast weight value and a scalar added to each sum, the scalar, in this case, the brightness value.

Figure 3.5: F1® 22 adapted contrast and brightness result

This adaptative solution has been tested with multiple video game interfaces and, as seen in Figure 3.5, the interface elements are now better contrasted with the background and the result can be used to obtain more precise results in the detection of edges and contours.

3.2.2. Adaptative noise and blur filtering

Noise can be defined as unwanted or undesired interference in a sound. Noise in an image is more related to random variations in the signal, brightness or colour, caused by issues with the image quality or processing steps.

In the case of Cogito, most game interface elements have images that are not related at all to the element itself or have backgrounds or dynamic elements that can severely affect contour or edge detection. To avoid future problems, after the brightness and contrast process, a blur filter should be applied to ensure the detection of only the important elements. But not just any blur filter can be used, since for proper adaptability, this filter must be correctly adjusted to each type of image.

The adaptative noise and blur filtering pipeline works as follows:

1. Once obtained the result of the adaptative contrast and brightness process, a new contrast and sharpness measurement is needed. This time more precise than the first

one due to the fact that the noise and blur filter is a more complex and delicate process. Too much blur filter can completely spoil the image or information for contour and edge detection can be lost.

2. The input image in RGBA is converted into gray-scale, this time to compute the image's histogram. This histogram is flattened and normalized by dividing it by the sum of its elements to ensure the correct distribution of pixel intensities.

   The histogram represents the distribution of pixel intensities of an image and its equation is defined as follows:

$$H(i) = \frac{n(i)}{N} \tag{3.1}$$

   Where $H(i)$ is the value of the histogram at bin i, $n(i)$ is the number of pixels of the image with intensity i, and N is the total number of pixels. In this context, bin is defined as a range of values between 0 to 255 used to group data for analysis.

3. Once the histogram has been normalized, the contrast of the image is computed by calculating the standard deviation of the gray-scale histogram pixel values. The contrast is calculated as the square root of the sum of the normalized histogram multiplied by the squared difference between each bin and the mean of the gray-scale pixel values.

   The contrast equation is defined as follows:

$$Contrast = \sqrt{\sum (p(i) - mean)^2} \tag{3.2}$$

   Where $p(i)$ is the probability of intensity level i of the image, and mean is the average intensity of the image. This equation calculates the standard deviation of the image pixel intensities, a measure of contrast.

4. Image sharpness is also a factor to consider when choosing the best blur filter, due to the fact that if an image has too little sharpness the blur filter could cause even more loss of information. In order to obtain the sharpness of an image the Sobel and Laplacian algorithms have been used.

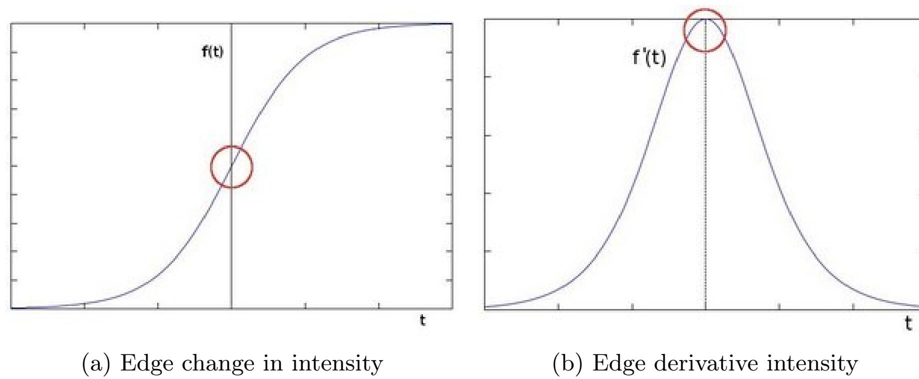
   - Sobel can be described as a calculation of the derivatives of an image. Edges can be defined as notorious pixel intensity changes, a good solution to find those changes is by using derivatives (Bradski and Kaehler (2008)). In a graphical way, an edge is shown as the change or jump in intensity and the derivative can help to detect these changes easily (Figure 3.6).

     But most importantly, the second derivative of an edge is zero, so this observation allows the tool to better detect edges in an image. (Figure 3.7).

   - The Laplacian operator is an extension of the Sobel operator. It is the one used in this project to detect the sharpness of an image using the second derivative for edge detection (Gonzalez and Woods, 2017). It is applied in both dimensions as it is a 2D image.

     The Laplacian operator for a function (image) $f(x, y)$ is defined as follows:

$$\Delta^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.3}$$



(a) Edge change in intensity (b) Edge derivative intensity

Figure 3.6: Sobel intensity graphic (Bradski and Kaehler, 2008)

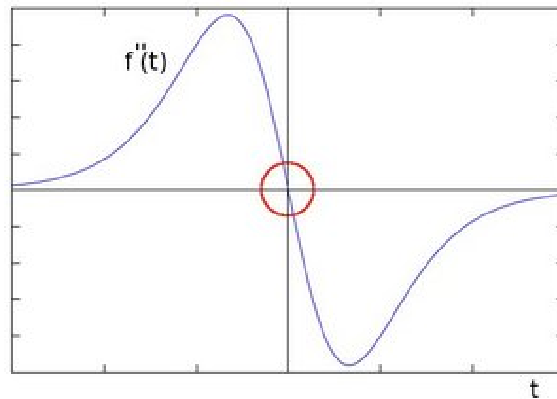


Figure 3.7: Second derivative of an edge (Bradski and Kaehler, 2008)

By calculating the variance of the Laplacian response over an image, a measure of the overall sharpness of the image can be obtained. The variance of the result of the Laplacian operator determines the sharpness of the image, where a high variance means a high sharpness of the image.

5. Once the contrast and sharpness values of an image have been obtained, depending on whether these values are too high or too low, the different blur filters to be applied to the image are chosen. The filters used are the Gaussian, Median or Bilateral filters:

   - **Gaussian Filter - High contrast**

     A linear filter used in digital image processing for noise reduction and smoothness. A linear filter operates on an image by making a weighted sum of the pixel values surrounding that pixel, these weights are defined in this case by a Gaussian function that gives more weight to the near pixels and less to the far pixels, which produces a smooth effect in the image (Gonzalez and Woods, 2017). The Gaussian function is defined as follows:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.4}$$

     Where $f(x)$ is the value of the function at the point x, μ is the mean of the distribution and σ is the standard deviation.

In the case of image processing, the Gaussian filter uses a kernel, a small matrix, to operate on the image using a 2D version of the Gaussian function:

$$G(x, y) = \frac{1}{2\pi\sigma^2} * e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.5}$$

Where $G(x, y)$ is the value of the Gaussian function at point (x, y) in the kernel, and σ is the standard deviation of the distribution, a large sigma value will result in a higher smooth filter applied to the image. This result is applied to the corresponding pixel in the image.

This type of filter is applied to images with high contrast. Since such images are sufficiently contrasted, it is only be necessary to reduce their noise and smooth unnecessary edges (Figure 3.8). The result may not be very significant to the human eye but when it comes to contour and edge detection it is a great help.

(a) Contrasted F1® 22 menu (b) Gaussian blur F1® 22 menu

Figure 3.8: Gaussian filter application on contrasted image, the entire image is blurred in order to obtain the most prominent shapes in the image.

- **Median Filter - Low contrast, low sharpness**

  A type of filtering technique used for noise removal from images, this type of filter is crucial in the image processing field since it preserves most edges during noise removal (Gandhi et al., 2020). This filter goes through each pixel of the image and replaces it with the median of its neighboring pixels, located in a square neighborhood around the evaluated pixel (Szeliski, 2010).

  The median filter will be applied to images with low contrast and low sharpness, as these types of images do not have enough sharpness at the edges, only those edges should be preserved and unnecessary noise removed (Figure 3.9). This filter will be applied sparingly because it can completely spoil the image due to its low contrast and sharpness.

- **Bilateral Filter - Low contrast, high sharpness**

  The Bilateral filter is a type of filtering technique similar to the Median filter, focused on smoothing an image while preserving most of the edges.

  It differs from the Median filter in that it follows a similar process to the Gaussian filter, replacing each pixel with a weighted average of its neighbors (Tumblin et al., 2009). These weights have two components, the first of which is the same weighting used by the Gaussian filter. The second component takes into account the difference in intensity between the neighboring pixels and the evaluated one (Szeliski, 2010).

(a) Contrasted F1® 2019 menu (b) Median blur F1® 2019 menu

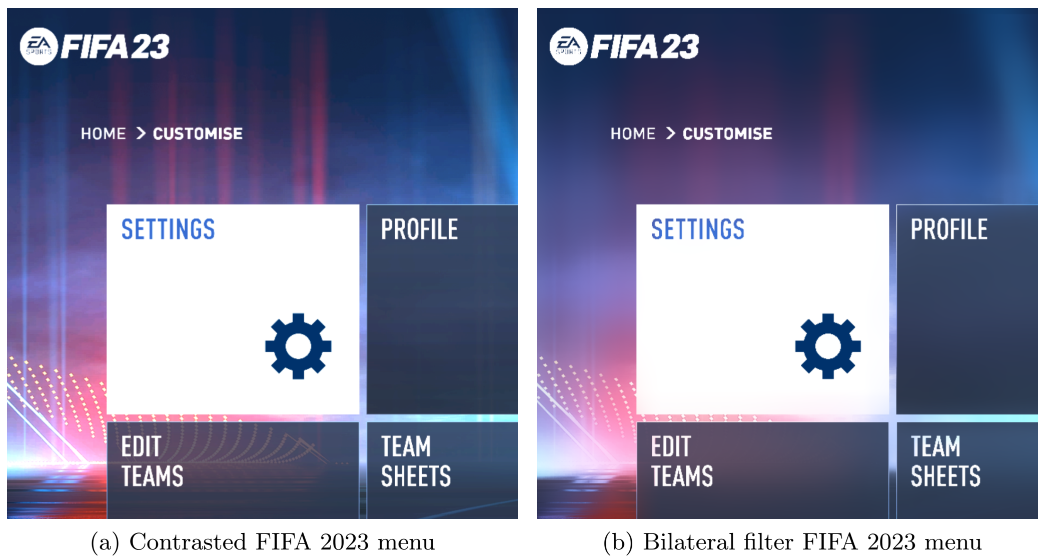
Figure 3.9: Median filter application on contrasted image, a minimal blur to reduce the noise in the image is applied. This modification of the image results in smoothed parts, such as the face of the figure.

This type of filter is much more complex than the previous ones and is used for low contrast images that retain a high sharpness value. As it is a more complex filter, its computational cost is also higher, which reduces the performance of the tool, although not significantly. The Bilateral filter applied to an image helps to soften specifically noisy parts of the image, while preserving edges and contrast elements (Figure 3.10).

3.2.3. Edge detection

As can be seen in the sections of this chapter, each part of the process is crucial to ensure a clear and legible result. The edge detection part is no exception as it helps greatly in detecting screen elements such as buttons or menu bars and, with the contrast and blur filters applied, the image is ready for this step. As the Machine Vision book stands in its Chapter 5: Edge Detection (Jain et al., 1995):

> *An edge in an image is a significant local change in the image intensity, usually associated with a discontinuity in either the image intensity or the first derivative of the image intensity (Figure 3.11). Discontinuities in the image intensity can be either (1) step discontinuities, where the image intensity abruptly changes from one value on one side of the discontinuity to a different value on the opposite side, or (2) line discontinuities, where the image intensity abruptly changes value but then returns to the starting value within some short distance. However, step and line edges are rare in real images. Because of low-frequency components or the smoothing introduced by most sensing devices, sharp discontinuities rarely exist in real signals. Step edges become ramp edges and line edges become roof edges, where intensity changes are not instantaneous but occur over a finite distance.*



(a) Contrasted FIFA 2023 menu (b) Bilateral filter FIFA 2023 menu

Figure 3.10: Bilateral filter application on contrasted image. This filter helps smoothing out the noise in specific parts while preserving edges such as the background and buttons in this figure.

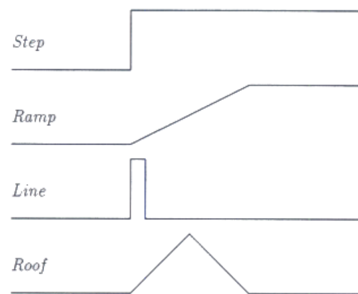


Figure 3.11: *One-dimensional edge profiles.* (Jain et al., 1995)

There are several algorithms for edge detection in an image such as the Sobel algorithm (Figure 3.6) which allowed the detection of edges by using the first and second derivative of the image. Although the algorithm used for this part of the project has been the Canny algorithm, due to the good results it produces.

Thanks to the work done above the Canny algorithm will work in a more adaptive way to any type of image but still it must also be adaptive in some way by itself, since the Canny uses mainly three parameters to work and these parameters must be adjusted to each type of image. These parameters are the aperture size and the upper and lower thresholds. This aperture size is set to 3 by default but the other two values must be adapted to the type of image in question. The upper and lower thresholds decide which edges are actually edges and which are not (Bradski and Kaehler, 2008).

$$cannyTh1 = max(0, (1 - sigma) * median) \tag{3.6}$$

$$cannyTh2 = min(255, (1 + sigma) * median) \tag{3.7}$$

In order to obtain the adaptive threshold values, a combination of conversion of the image to gray-scale and the median of the image pixels using the formulas 3.6 and 3.7 was used. In these equations the sigma value is set to 0.33 while the other two depend on its gray-scale median values, this process will be applied to each channel of the HSV image due to the possible loss of information when converting to gray-scale, by doing this on each channel and merging the results the tool makes sure that no information is lost and obtains a more accurate result.

Figure 3.12: HSV Multi-Channel Canny result

As can be seen on Figure 3.12, the combination of each HSV channel Canny results on a binary matrix containing the edges of the image. The importance of the brightness, contrast and blur filters explained above in performing the Canny algorithm will be appreciated below.

- **Gaussian Filter - Canny**

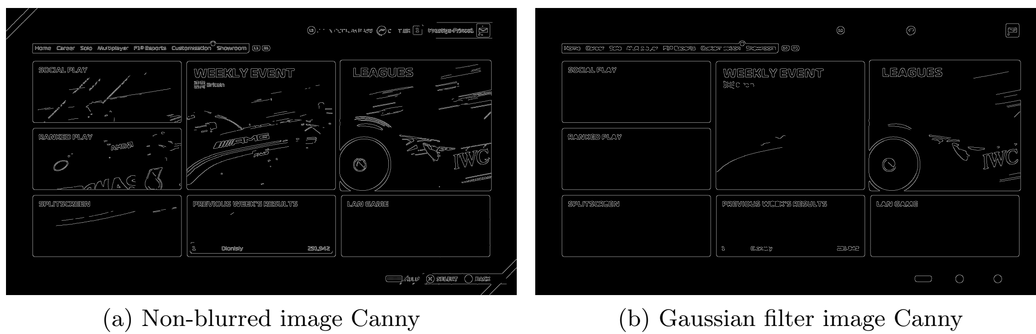
  As explained above, the Gaussian filter is used for high contrast images in order to remove unnecessary noise or edges, as can be seen in Figure 3.13, the Canny algorithm is able to capture most of the edges in a non-blurred image but maintains a considerable amount of noise, so it is necessary to apply a Gaussian filter in order to reduce such noise.

- **Median Filter - Canny**

  The median filter is used for images with low contrast and low sharpness, due to the low sharpness value it is applied in a reduced way to maintain as much edges as possible and reduce the noise, as seen in Figure 3.14.

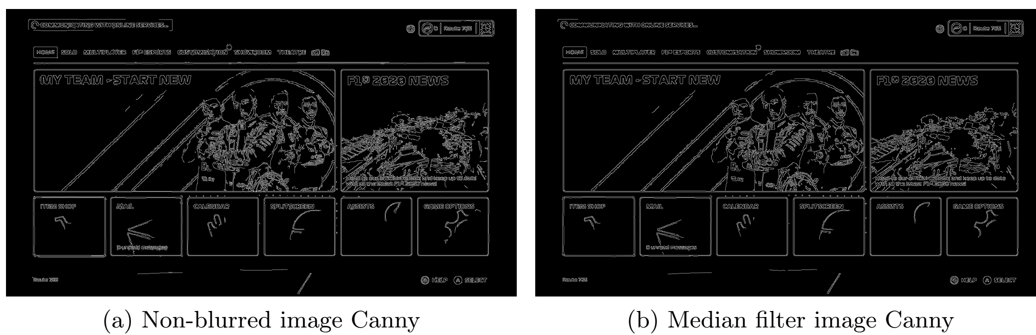
- **Bilateral Filter - Canny**

  The bilateral filter is used for images with low contrast and high sharpness. Images with a high sharpness value could indicate that unimportant backgrounds or elements stand out in the image. This filter is perfect to reduce this kind of noise and unnecessary edges. As seen in Figure 3.15 background noise is considerably reduced.



(a) Non-blurred image Canny (b) Gaussian filter image Canny

Figure 3.13: Comparison of Canny applied to a Non-blurred and Gaussian filter F1® 21 image. Contours and edges from inside each detected element are moderately reduced on the right after the filter is applied.



(a) Non-blurred image Canny (b) Median filter image Canny

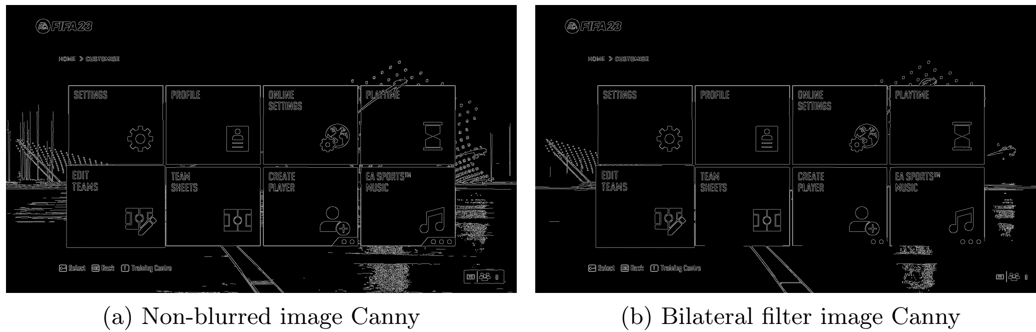
Figure 3.14: Comparison of Canny applied to a Non-blurred and Median filter F1® 20 image. The noise is slightly reduced for the tool to correctly detect the elements on the screen.

3.2.4. Contour detection

Contours are curves joined by all continuous points along a boundary that have the same color or intensity (Bradski and Kaehler, 2008). To correctly detect the contours of an image, some steps must be followed:

1. The image to be detected must be binary for greater accuracy. To do this the canny algorithm is used.
2. Based on the binary image, a contour detection algorithm is used.
3. Contours are simplified and filtered to keep only those that are closed or convex. (Figure 3.16).
4. The bounding boxes are obtained from the contours in order to have simplified shapes and thus obtain most of the buttons or interface elements (Figure 3.17).

This step is relatively simple thanks to the work done in the previous phases, allowing a more accurate detection. Throughout this process the tool has applied different brightness, contrast and blur filters in order to be able to use the edge and contour detection algorithms in the image.



(a) Non-blurred image Canny (b) Bilateral filter image Canny

Figure 3.15: Comparison of Canny applied to a Non-blurred and Bilateral filter FIFA 23 image. Most of the noise from the background is removed.

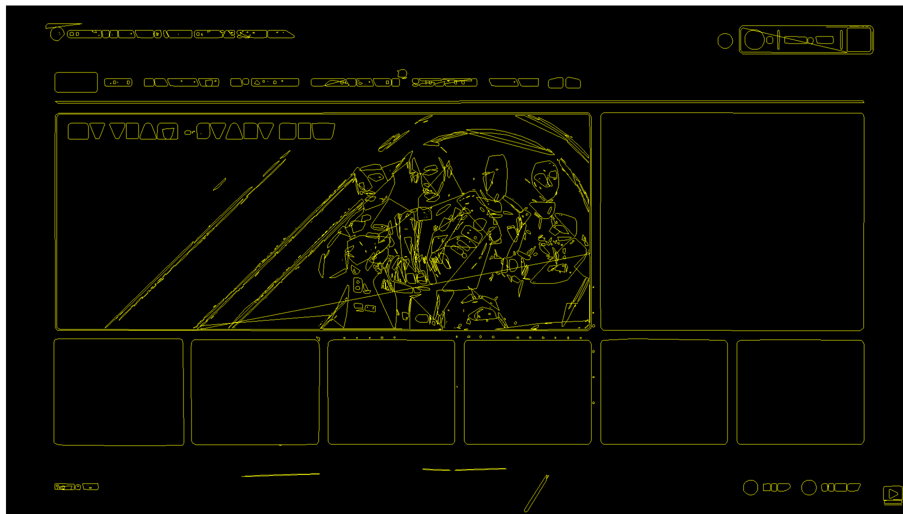


Figure 3.16: Filtered contours of a F1® 2020 image

At this point it is necessary to filter these contours by keeping only those that contain a meaningful element, these elements being text or icons. This is done because a meaningless contour is probably not part of a menu. By obtaining texts and icons the tool will then be able to classify them and then find structures of navigable menu elements.

3.2.5. Text detection

To detect text, an OCR machine learning model is used. After extensive testing, it has proven to be very effective and is able to detect text from a large number of different sources. Although it is a very feasible solution, it still requires further refinement which will be described below:

1. The machine learning model recognizes text word-by-word due to possible errors when detecting paragraphs or lines of text.

2. Once all the words of an image have been obtained, those that are not in horizontal orientation will be discarded. Then those words that are sufficiently close and aligned horizontally will be merged together and those lines of words that are close and

Figure 3.17: Bounding boxes of a F1® 2020 image

aligned vertically will be joined together (Figure 3.18). With this process the tool is able to detect paragraphs and lines of text.

3. Finally, the returned result will be a list of all the texts with their respective bounding boxes.

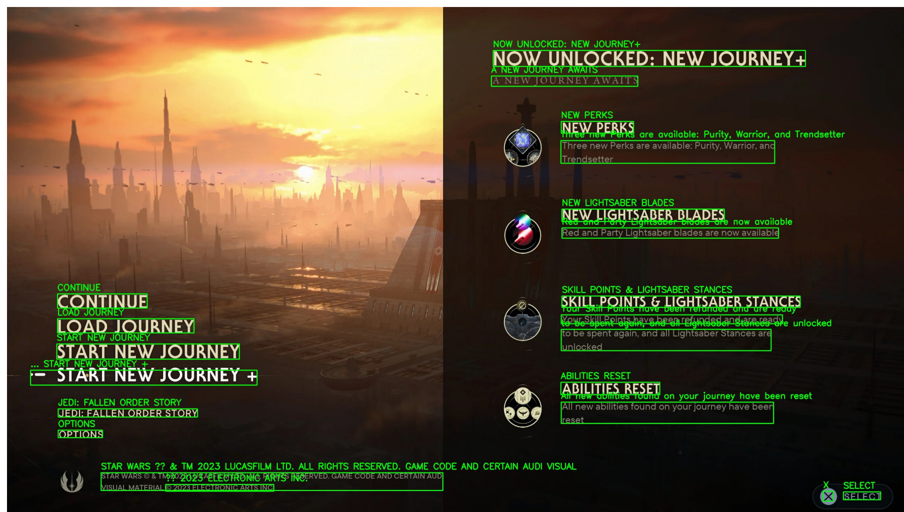


Figure 3.18: Text detected on a STAR WARS Jedi: Survivor™ menu

3.2.6. Icon detection

In addition to text detection, a general icon detection is also performed. For this purpose, a trained machine learning model is used to recognize elements that resemble an icon in a screenshot. This model has been used because it is possible that some buttons only have an icon without text and it is necessary to select it to know what it is, so it must be detected first.

This model is trained with a database of different images containing icons of any type,

all labeled with the same label. The model will receive an image and will detect in it the elements that have any resemblance similar to an icon.

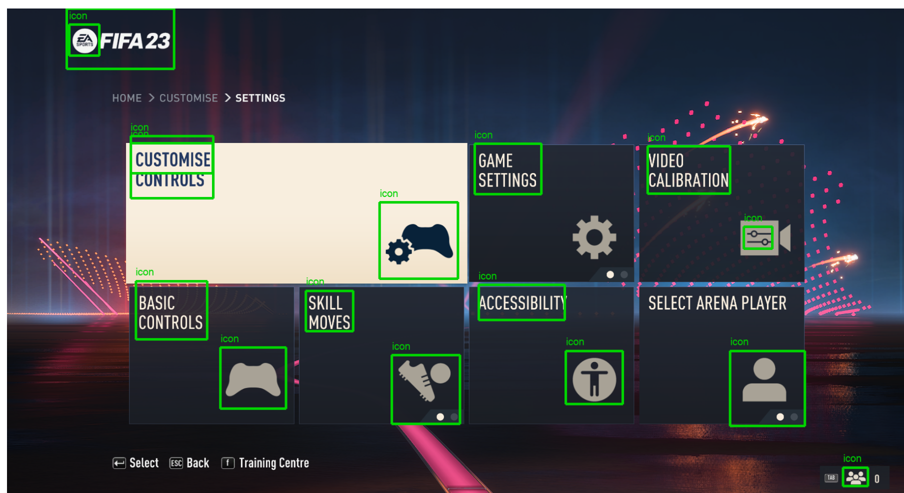
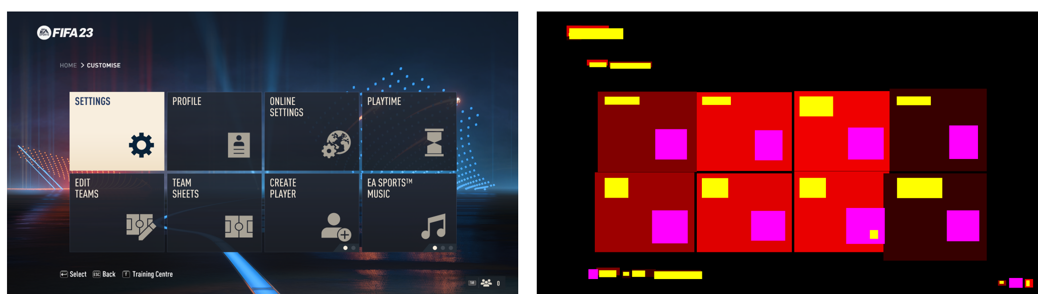


Figure 3.19: Icons detected by machine learning model on a FIFA 23 settings menu

As can be seen on Figure 3.19, the model is able to detect most of the icons in an image and also considers text as icons, since during the training process some of the icon images used contained text. But due to the previous text detection it is possible to discard those texts detected by the general icon model and keep those true icons.

Now that the bounding boxes of the contours as well as the text and icons of the image have been obtained, it is time to combine all the previous steps. The tool will merge both results in order to filter out the important elements of the image. For this purpose, those contours that contain a text or an icon are considered as relevant and because the contour detection could fail, it has been decided to keep those texts and icons that are not contained in a bounding box, while the rest of the contours will be discarded.



(a) FIFA 23 Settings menu

(b) Text and icon detection combined with the detected buttons on the image.

Figure 3.20: Text detection and icon detection performed on a FIFA 23 menu. Left being the original image and right being a visual representation of each kind of element detected.

As can be seen in Figure 3.20, the red chromatic boxes are the contours of each element, the yellow boxes represent the text and the purple boxes represent the icons. This result is used for the next step of the tool, classification. This is in charge of classifying the texts and icons of the data obtained. Once the elements have been classified separately, the tool will seek to group these elements in order to find a navigable menu.

3.3. Element classification

Now that information like the position of elements or their contents has been obtained, this part of the pipeline will focus on classifying these elements. This will not only into texts and icons, but also meaningful elements

3.3.1. Text classification

In the case of text classification, the tool mainly uses a dictionary of labels (Figure 3.21), where each label contains words or phrases associated with that label. The process to classify a text using this dictionary consists of going through each word of each label in the dictionary and making a comparison between the word and the text to be classified. This comparison will be based on word similarity and the confidence will be determined by that similarity.

LABEL exit	LABEL settings
quit game	options
game menu	settings
exit	configuration
leave game	local configuration
quit	game options
exit to desktop	player settings

Figure 3.21: Exit and Settings label dictionary examples

This process is applied to every text type entry of the previous phase (subsubsection 4.2.3.2). Text not classified correctly or with a confidence value below a defined threshold will be discarded. Figure 3.22 shows an example of text classification using this solution, using green boxes to surround classified elements, marking them with their corresponding label.

Although it has been mentioned that this solution does not need tedious maintenance, Cogito will still need to be maintained to keep the different labels in check with the video games to be tested with. Even so, maintenance will be minimal and this will be the only critical aspect to be updated. In the case of text classification, the label dictionary part of the tool is the one that needs the most maintenance, but it can be done quickly as it only means including a word or phrase in the label dictionary.

3.3.2. Icon classification

The process followed by icon classification is similar to that of icon detection (subsubsection 4.2.3.2), but in this case applied only to specific types of menu icons. The model

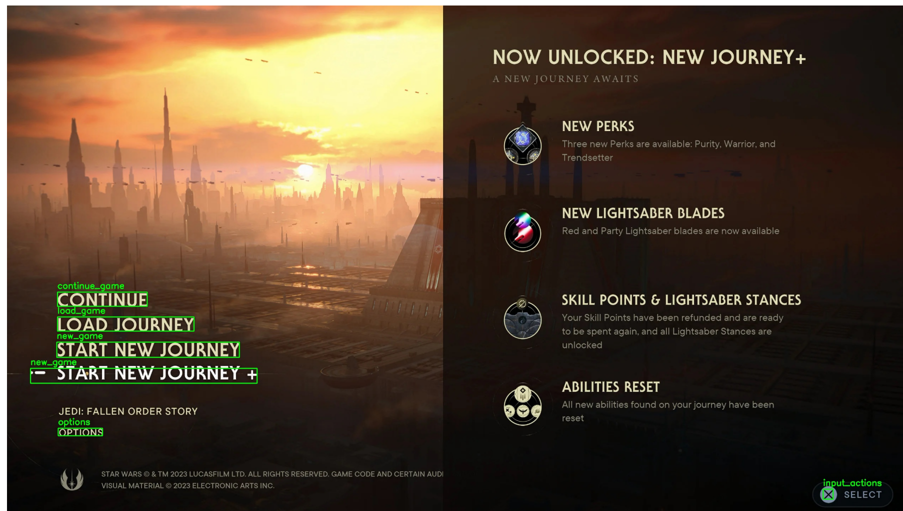


Figure 3.22: Classified texts on a STAR WARS Jedi: Survivor™ menu. The green boxes outline the detected texts and display the label classified on top of them.

has been trained with multiple examples to be able to recognise typical icons that appear in multiple video games such as the Settings, Accessibility or Controller icons. Each type of icon is a label for the model to train.

This solution, in combination with the icon detection result, can be very useful to specify which function a button has and thus be able to classify buttons that do not contain any text. Figure 3.23 shows an example of icon classification using the aforementioned model, using green boxes to surround classified elements on the same way that was done with texts, marking them with their corresponding label as well.

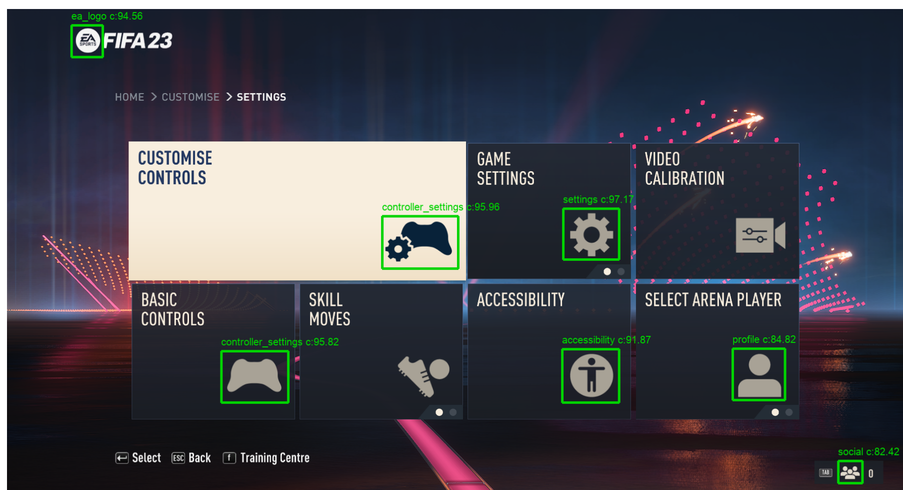


Figure 3.23: Classified icons on a FIFA 23 settings menu. The green boxes outline the detected icons and display the classified label and confidence on top of them.

3.3.3. Meaningful element groups

Now that most of the elements of the screen have been obtained and they have been classified, it is time to check various aspects of the elements in order to know in which

parts of the screen can the tool navigate through. The steps to perform are the following:

- Element grouping detection (subsubsection 3.3.3.1) - Groupings of elements or buttons will be detected following Gestalt's principles to differentiate between groups (section 2.1).

- Highlighted button detection (subsubsection 3.3.3.2) - Inside each of these groups, the selected button must be detected in order to properly know where it is located during navigation.

- Finding navigable menus (subsubsection 3.3.3.3) - It is necessary to know which groups of elements are truly part of the menu interface and can be navigated through. This case will help to establish the navigation's entry point and to know if we have reached the desired destination.

3.3.3.1. Element grouping detection

The objective of this stage is to obtain the groups of elements that form an aligned structure because they are more likely to belong to a video game menu. This approach has been decided due to the current state of video game interfaces, which are largely based on Gestalt principles and maintain adequate accessibility for players (section 2.1).

The tool will search for elements that are aligned both horizontally, vertically and in the form of grid, in order to obtain the possible lists of horizontal, vertical or grid buttons that may be on the screen. Following the Gestalt principle of proximity, the tool assumes that the elements that are close and aligned will belong to the same group. Once the groups of horizontally or vertically aligned elements are found, the tool will search if there is a relationship between these groups, in order to find a grid of elements as well.

When talking about elements, the tool will mainly seek to group texts and icons that are aligned. These elements may or may not be actually related and for this the tool must be able to keep only the correct groups of elements, that is why the tool is required to have a way to find navigable menus.

3.3.3.2. Highlighted button detection

The detection of the selected button is a crucial part of the tool because knowing which element has the focus greatly aids navigation. If we look at video game interfaces, the selected button usually has a different contrast level than the rest or a different color, so this information is very useful when it comes to finding the different button.

To do this, the tool uses an algorithm based on histogram intersection comparison. When getting the groups of buttons the tool acquires the histogram of each of the buttons and compares them with the rest using the intersection algorithm. If a histogram has a similar range of colors or contrast its intersection will be greater while if they are totally different the intersection will be minimal (subsection 3.2.2). By obtaining the intersections between every element in a group the tool is able to know which element is selected because it will have a much smaller intersection than the others.

3.3.3.3. Finding navigable menus

In order for the tool to be able to know if there is a navigable menu on the screen, it is necessary to use the groups of aligned elements and keep only those groups that have a meaning.

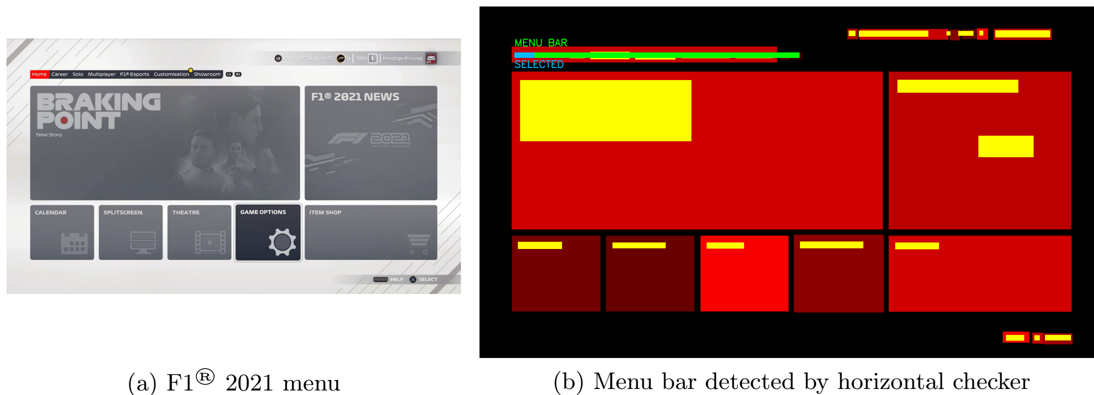
That is why the tool uses checkers to identify those groups, which are in charge of checking that the different groupings of aligned elements contain elements classified with labels belonging to a menu interface. Checkers help the tool find the correct groups of elements and the tool will use them to find navigable menus or menu components such as menu bars.

These are the following types of checkers:

- **Horizontal and Vertical Checker**

  These checkers are mainly in charge of finding in the horizontal or vertical aligned groups obtained previously if they comply with the rule previously mentioned, that the elements of said group have specific labels. This rule is used so that these checkers only store groups of elements that can belong to an interactive menu.

  They can work well when looking for menu bars or profile bars, as such menu items usually have the same type of labels in any game.



(a) F1® 2021 menu

(b) Menu bar detected by horizontal checker

Figure 3.24: Horizontal checker for menu bar detection on a F1® 2021 menu

  As can be seen in Figure 3.24, the menu bar is detected correctly and even includes which button is currently selected.

- **Grid Checker**

  This type of checker uses the grid element detection mentioned above which combines both horizontal and vertical groups to find a grid which correctly is a menu interface. Uses the same rule as the previous checkers, if the elements share the same or some of the labels belonging to a menu interface.

After the process of data classification and checking, the tool returns all the collected information to the navigation application for it to use. This information includes texts, icons, contours and groups of menu elements with their selected element. Figure 3.25 represents a visual example of the final result.

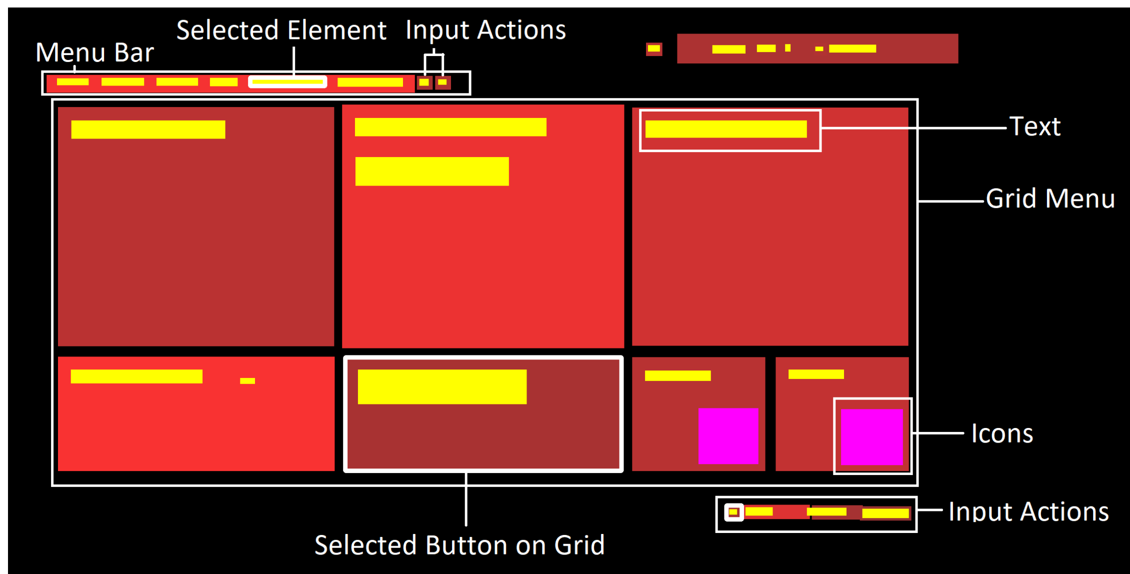


Figure 3.25: Example result of all the final information gathered by the tool.

3.4. Choose the element to interact

The Cogito application is able to run a game, take screenshots, send those screenshots to the Navigation pipeline and based on the results returned by the detection and classification workflows it will perform some input actions.

It is important to highlight that the navigation part was not the main focus of this project. Still, its design is briefly explain as how it has been thought,

Once the identification and classification of elements is done, now the navigation starts. It uses the data obtained by the previous steps to perform an automatic navigation through the entire menu of a video game, selecting all the menu buttons. This approach is a simple start to demonstrate the potential of the tool.

From the data obtained, the input made by the tool changes according to the type of navigable menu in which it is located. If it is horizontal or vertical it will move along that axis and if it is a grid it will move along both axis.

The tool will search for the destination element on the menu and if it does not find it, it will search for elements with the same label as the destination. The tool considers that it has reached the destination if it has selected the element or if it has found it in the screen. In case the navigation gets stuck or does not reach the desired destination it will stop and declare the test as a fail.

3.5. Conclusion

Throughout this section the pipeline of the Cogito tool has been described. This is divided into multiple parts. The work done has mainly focused on the step regarding identification and classification of elements, which forms the entire analysis part of the tool and its the one explained with more detail.

The navigation application sends a screenshot of the current state of the game to the image processing workflow for analysis. This workflow starts by adjusting the brightness and contrast of the input image, and then adaptive blur filtering based on sharpness and contrast is applied to improve edge and contour detection.

To correctly detect the edges, the Canny edge detection algorithm is applied to each HSV channel of the gray-scale image using adaptive parameters based on the median values of each channel. Once the edges are detected, the contour detection algorithm is performed and the result is simplified into its bounding boxes. The text and icons in the image are then detected by machine learning models and the results are stored in a list that retains only the contours containing text or icons.

With all the data collected, text and icons are classified into labels using a label dictionary matching algorithm and a machine learning object detection model. Then all the sorted data is used to find meaningful element groups in the image such as menu bars, the selected button or the current menu, grouping the elements based on their alignment and checking their labels.

Finally, all the classified data, the selected button and meaningful element groups are returned for the navigation part to use, performing input actions to navigate through all the elements in the menu.

In the next chapter it is explained how the tool's architecture is structured and implemented.

Chapter 4

Development

This chapter describes the development process and the decisions taken to develop the Cogito tool, to recognise and classify the elements of a screenshot for menu navigation in any video game. Throughout this chapter, the whole evolution of the project will be described. This project has followed an incremental development based on prototypes, starting with preliminary research and leading to the final solution. This chapter will explain in detail the problems throughout the development process, architecture and configuration of the tool.

4.1. First research and prototypes

The research part of the project started months before the project members joined the Electronic Arts team. This research helped the team to define the scope of the project and to find functionalities in existing tools to apply to Cogito. The aim was to find ways to detect on-screen elements as well as to investigate technologies similar to the proposed solution.

4.1.1. Tesseract text detection

Since most buttons in menu interfaces include text to describe their meaning or functionality it was decided to investigate text detection. Tesseract is one of the most common and reliable open source OCR (subsection 2.3.1) therefore, a prototype was implemented using this solution with OpenCV to become familiar with this technology.

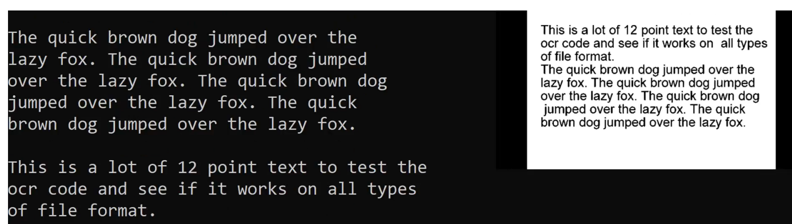


Figure 4.1: Tesseract text detection prototype simple example

As can be seen in Figure 4.1, Tesseract seems to correctly recognize text in a plain text document. But as seen in Figure 4.2 although it correctly detects some text, it still has difficulty detecting text in an image with a dynamic background or unusual fonts and may cause strange text detection.

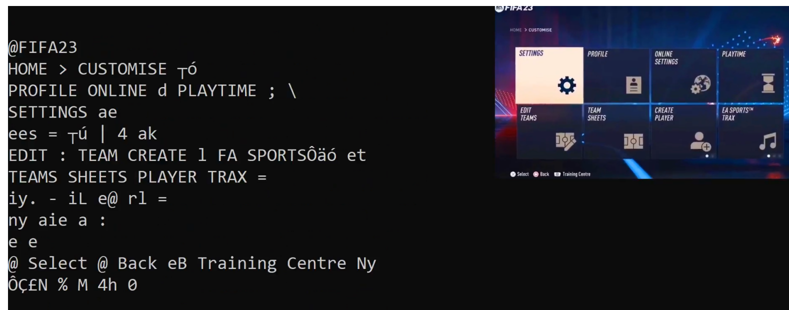


Figure 4.2: Tesseract text detection prototype FIFA 23 menu example

4.1.2. GameDriver functional testing

In order to fully understand the already existing tools related to functional testing, an Unity project with the GameDriver API was created to test its capabilities. As explained before in subsection 2.2.1, this type of testing API requires to be implemented inside the engine or game itself and for that reason it is not an adaptative solution for interface navigation, at least for games that do not use the Unity engine.

Even so, the project members were able to find interesting implementation details of this API to use for the Cogito tool:

- Waits, delays must be implemented in the navigation to allow the game to load or process the input. The tool may wait for a specific item or label to appear or for the screen to change significantly.
- By having internal access to the application structure, actions can be performed on its elements. As the tool will not have internal access to the application, it will be able to create its own artificial structure based on the visually detected elements.

4.1.3. Sikuli visual testing

Sikuli is an open source tool used for automated visual testing as described in subsection 2.2.2. Since Cogito is intended to work around multiple different titles, it was necessary for the team to investigate this tool.

A test case was conducted to demonstrate the accuracy of the tool, as Sikuli used template matching for inputs and waits, so it was very easy to use, but this feature is not useful for a tester, as taking screenshots of certain items in a menu to navigate through it can already be time consuming (Figure 4.3).

From this research the team drew the following conclusions:

- The research on Sikuli led to the decision to create a tool based on visual testing, as it was the closest approach to the final idea.

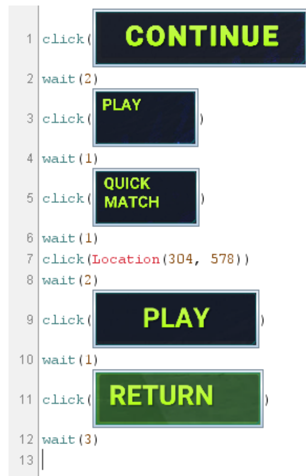


Figure 4.3: Sikuli test example

- Sikuli's easy-to-use Python API gave the team an idea on how to structure the scripting language for simple actions like *"Go to Settings"*.

4.1.4. Amazon Web Services (AWS)

As a project constraint, Amazon Web Services technology were used for everything related to machine learning subjects. Using these technology will be useful for two things: detecting the data and classifying it. The services used in the project are the following ones:

- **Rekognition**

  An AWS technology focused on being user-friendly, while being a powerful tool. Rekognition allows to train models based on deep learning to classify images, detect objects inside them, perform facial recognition and even video analysis. The most interesting capabilities related to this project are two of these. First there is text detection, which is an already trained model with the most modern techniques applicable to the OCR tools. It can detect correctly all kinds of texts fonts and sizes, even stylized ones, as Figure 4.4 shows.

  The other important capability is the Custom Labels service. Rekognition allows users to train models without needing the user to have very low level knowledge. The models trained can be of 2 categories, depending on its function. If images without any position labeling are used to fed to the model, the result will be an image classifier. However, if the images have been labeled marking specific regions with labels, then the result will be an object detector.

  This service requires a dataset and its proportional division between test and validation datasets. Iteration throughout these models is fast as the fitting is done automatically. In addition to that, the dataset used for these models can be used in Sagemaker with some variations, since both services have them stored on another service they have in common, Amazon S3.

  One of the biggest advantages of Rekognition is its cutting edge technology, allowing the user to provide limited amounts of data and still have good accuracy on the

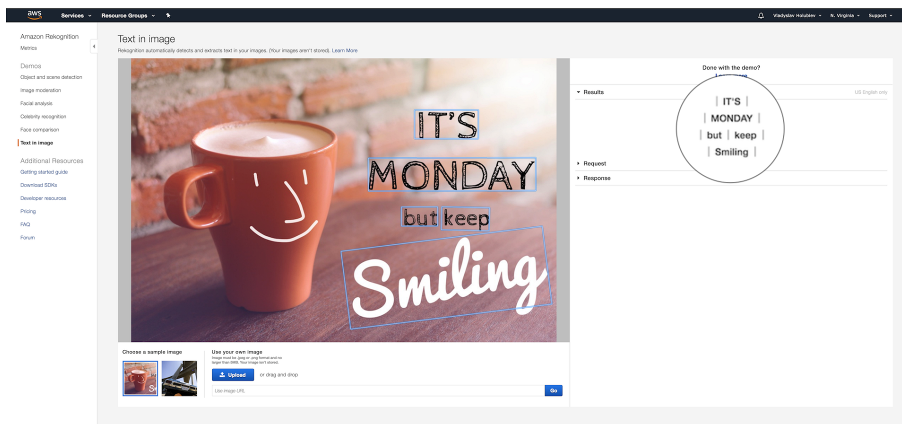


Figure 4.4: Example of Rekognition text detection.

models. For example, the minimum recommended training images per label is 25.

In order to know if models are performing correctly, each training returns some metrics that help to understand what is happening:

- Precision: Predicted true positives over all true model predictions. That means, if the model detects incorrectly instances of an element in an object detector, the precision would be lowered.
- Recall: Predicted true positives over all relevant cases, that meaning only are taken into account which predictions are true positives over the actual true positives in an image, but not also over false positives.
- F1 score: The harmonic mean between precision and recall.

▪ **SageMaker**

A fully managed machine learning service. It provides an API that goes from building a specific model to deploying it for use. Unlike Rekognition, it supports multiple machine learning algorithms to train all kinds of models. Gives the user access to its neural networks and the possibility of tuning them to get the best results. Since the user has more control over the training, machine learning knowledge is required to obtain good results.

Text classification models are also supported. These models can be trained either building a model from scratch, or applying transfer learning over other popular text classification models.

Each algorithm has its own success metrics. In the case of object detectors or text classifiers, the one used is Mean Average Precision (mAP).

▪ **Lambda**

A service that offers an environment to execute back end code on a server without needing the user to manage it. Server calls can be made to communicate information in an easy way, using payloads that can contain bytes of an image, for example.

This technology combined with a continue integration system can provide an easy way to keep code updated on a cloud service. It provides a safe and isolated environment able to manage events or changes, and also an easy integration with the rest of AWS services.

4.1.5. Prototype of a machine learning model

The team focused on learning about the different machine learning services provided by Amazon Web Services and concluded that an object detection model was needed to recognize elements on a menu screenshot.

The first model was trained using SageMaker's object detection service, it was trained using a dataset of 100 images of F1 menus, since there was not that many data to collect for training, data augmentation techniques were used. These techniques included changing the images hue, contrast and orientation.

As can be seen on Figure 4.5, the model shows a very low accuracy and also was wrongly trained due to the use of only one label, the button label. However, this first attempt helped to automate model training on SageMaker and Rekognition, a small tool was created to train models in these services locally.

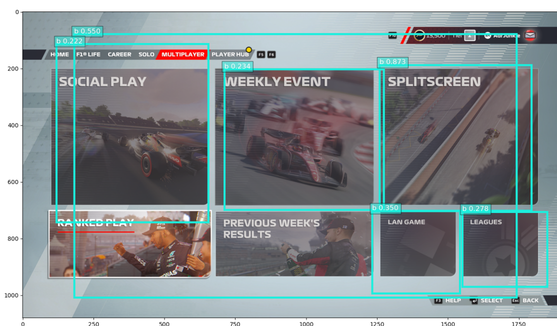


Figure 4.5: SageMaker first button detection model

Although it would not change much to use a larger number of labels for different types of buttons as the accuracy would remain quite low without overfitting the model. The problem was mainly the amount of unnecessary information contained in the buttons, either images of cars, people or objects out of context or icons with no understandable meaning.

A general button detection model was not possible because of all the different types of button styles from every game. So other alternatives for detecting elements had to be found.

4.2. Iterative development

Once the prototypes and research were done, the development for Cogito started. This process can be divided into different iterations where the different parts of the tool were implemented and improved.

4.2.1. First Iteration - Text detection

With the object detection model problems acknowledged, the development started by focusing on text detection, since most of the buttons contained text indicating their mean-

ing. Two machine learning text detection models, AWS Rekognition and AWS Textract, were tested. Rekognition was shown to work with various types of fonts of different titles, while Textract was a model focused on document text detection, so the first text detection model was chosen. The process to use the OCR is described on subsection 3.2.5.

(a) Example of AWS Textract, almost none of the text was detected.

(b) Example of AWS Reckognition, most of the text was correctly detected.

Figure 4.6: Comparison between AWS text detection services on Madden.

As can be seen in Figure 4.6, Rekognition's text detection outperforms Textract's in video game cases, detecting all the relevant text of the screen.

Although Tesseract OCR was investigated at the beginning of the project, its results were not satisfactory due to font detection limitations. Additionally, it had the same problem as AWS Textract, it worked better with documents than with dynamic interfaces.

4.2.1.1. Text detection limitations and improvements

Rekognition text detection requires a byte-encoded image as input and returns a dictionary object, which indicates attributes about the detected texts, such as position, detected words, confidence and an unique id for each phrase. It should be noted that texts can be detected as separate words or as phrases. However, it has occurred that words that were part of the same phrase were not always grouped correctly. In addition, words were only grouped horizontally, but not vertically. For this reason, word detection mode is used, and an internal adjustment is made that forms groups based on the vertical alignment of words, and the distance between words, based on their font size.

One problem encountered in testing was the *one hundred word limit*: Rekognition limit restrict its detection to a hundred words per call. This caused some images exceeding this amount to have missing words at the bottom of the screen, because word detection goes from the top left position of the image to the bottom right, reading words from left to right and top to bottom. This was solved by making several calls, adding a black box to the regions before where the last word was detected in the previous call and combining the results from the different calls. Thanks to this solution, the tool is able to detect all texts correctly and thus also classify them correctly.

4.2.2. Second Iteration - Image processing

Although the use of models that detect any type of buttons was discarded, an attempt was still made to simplify the data for both training and validation, because if what

hindered the model was unnecessary information, keeping only the necessary information would improve the results. This led to the creation of an image processing module, which simplified a screenshot of an interface to red and yellow boxes, signifying contours and texts respectively. Due to the great results provided by this solution, the button detection model was ultimately discarded and the project focused into improving the image processing.

The objective of this image processing is to obtain those elements that are in the foreground of the image, performing edge and contour detection techniques on images with specific blur and contrast filters. The whole final process is explained with detail above (section 3.2). Even so, after the whole process some cases may occur in which the elements still do not have their contours correctly detected (Figure 4.7).

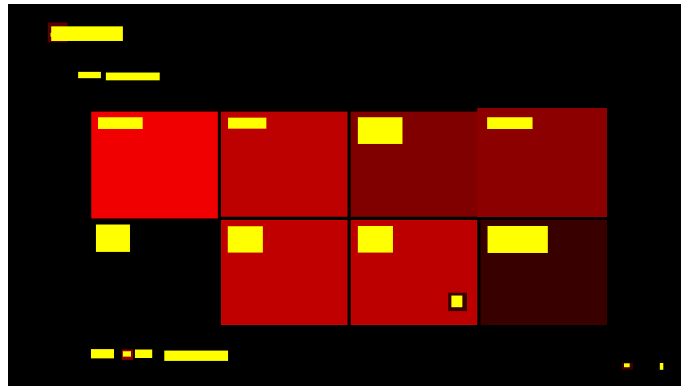


Figure 4.7: Missing contours in a FIFA 23 settings menu

4.2.2.1. Missing contours solution

The detection of bounding boxes is not perfect, so a missing contour detection algorithm has been developed that based on alignment creates a new contour for the contourless text or icon. The algorithm replicates the bounding boxes of near aligned elements to fit the missing space. If the element is not aligned with any other, it is assumed that it does not belong to a button list or anything like that so no contour is applied to it.

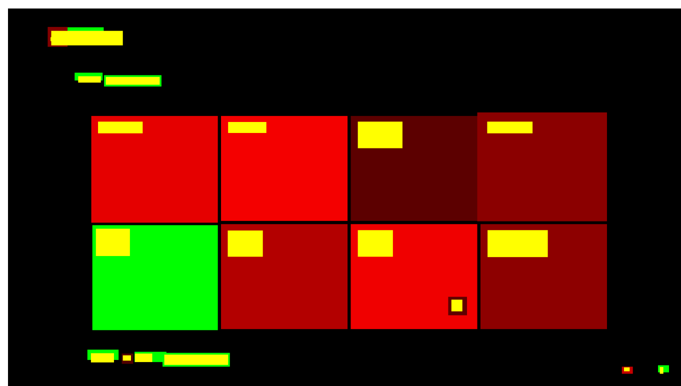


Figure 4.8: Artificial contours for missing contours detected

4.2.3. Third Iteration - Element classification

Once the button tiles were correctly detected, the next goal was to be able to classify each element with a meaningful label that can be used in future steps for navigation. Elements are shapes reassembling buttons on the screen that can contain either a text, an icon, or both. Multiple approaches were tested for this task.

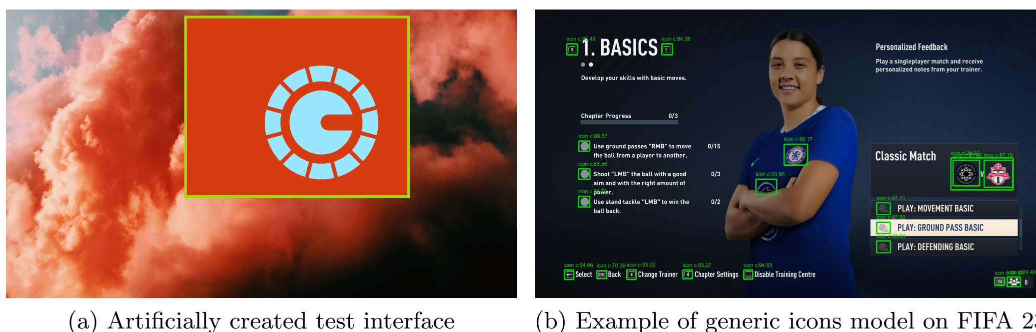
4.2.3.1. General icon object detection model

Using the AWS Rekogntion custom labels service, several models were trained with the objective of detecting just the icon positions, but not its meaning, as it was described on subsubsection 4.2.3.2. This was done in order to facilitate the image processing step, since the general positions are needed to know what is a button and what is not, even when the icon is too abstract for the models to determine its meaning with the next model.

The idea was using multiple screenshots from video games labeling all menu icons under the same label. However, in order to achieve a good accuracy, models need from regularization data to avoid overfitting, a problem that would severely affect the usefulness of this mode.

The dataset used for this model includes not only video games screenshots. but a combination of screenshots and artificially created menus with generic icons (Delapouite and Lorc, 2022) over backgrounds. Over these artificial menus, multiple things were tested, for example changing its orientation, transparency, color and background. The final dataset is composed of 44 game screenshots and 167 generated images for training, 9 game screenshots and 46 generated images for testing.

A small script was made in order to generate this artificial menus with variations and obtain their positions to label them automatically with the generic label and its position on the screen. The reason for doing this is that the model must be an object detector, so it needs to be trained with positions so that later it can also detect icon positions. Except for the modifications to the transparency, all other changes were satisfactory improvements, resulting in a highly accurate model.



(a) Artificially created test interface (b) Example of generic icons model on FIFA 23

Figure 4.9: Example of generic icon's dataset and its use on FIFA 23

In the previous figure Figure 4.9 (a) shows one image from the dataset with the previously mentioned modifications, whereas (b) shows how generic icons are detected with a confidence over 80%, without filtering the icons not needed.

4.2.3.2. Icon detection model

This model is based on part of the dataset used for the generic icons model (see subsection 3.3.2). However, generic icons can not be used for this approach, so regularization must be done with specific icons, which causes on a larger sized amount of icons used. This is due to finding specific icons is a harder task which will require more effort than in the general model.

In addition to that, this model includes multiple labels for the most common elements that have the same meaning between different video games genres. For example, some labels used are settings, audio, accessibility or social.

It is important to note that this model will not be used on icons which meaning varies depending on what genre you are playing. An example of this issue would be an icon of a house. In the case of a generic game, this icon usually means "return to the main menu". However, in a racing game, a house icon can be identified as "garage" or "customization".

Since icons that are from the same label can appear in different shapes, multiple sub-labels for the same label are used when needed. An example of this would be the most common icons for sound, a speaker icon, and a headphones icon. If only a label was used for each kind of icon, the model would fit worse to the already limited train icons, which would result on worse prediction scores.

For both generic and specific icons, a SageMaker model was trained as well using the ResNet-50 Convolutional Neural Network (see subsection 2.4.1)

4.2.3.3. SageMaker text classification

Text classification has been tried with multiple approaches. Three kinds of models were tested and compared to know which one was able to give better predictions. Furthermore, training a model from scratch is very challenging, and needs of large amounts of data to be reliable. However, this was tried for research purposes at this iteration.

The accuracy of this model is very low, having the mean average precision (mAP) fluctuating out of control. Also it is not able to detect similar semantic relationships, having a hard time to classify even variations of words with the same roots under the same label.

The second approach was training over an already trained model. SageMaker allows transfer learning for text on a list of models. In the case of text, it supports BERT model variations. BERT (Bidirectional Encoder Representations from Transformers) provides dense vector representations for natural language by using a deep, pre-trained neural network with the Transformer architecture (Devlin et al., 2019). Although its accuracy is more stable compared to the previous approach and part of the text is correctly labeled, it still happens that on the same label, successfully classified text scores can vary greatly. This is a huge problem, causing the classification to be unreliable, since a low score can mean either a correct or incorrect classification at the same time.

The third approach was using a revolutionary technology called Zero-Shot learning. It consist on models trained on natural language processing techniques with the difference of them being able to predict labels that were not specifically trained with. This model by itself is versatile, allowing to detect any existent labels if enough information is used. A

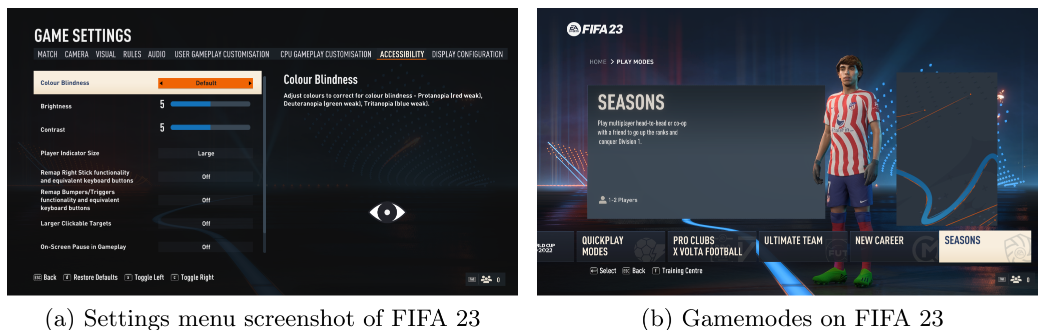
disadvantage that it has compared to previous models is that it does not allow tuning or transfer learning using SageMaker, so the results are limited to what the model offers by default.

In the end, due to the unreliability of these solutions, it was decided to implement a dictionary-based text classification system, which is a more robust solution than the previous ones.

4.2.3.4. Dictionary text classification

Dictionaries appear as an attempt to improve text classification. Each label has a list with the most common words for each label gathered from multiple titles from Electronic Arts. This guarantees phrases being correctly labeled when detected, but at the same time it lacks adaptability. The use of dictionaries is explained on subsection 3.3.1.

Something that is done to increase the chances of a correct classification is detecting partial match instead of the full word. The reason to do this is that the OCR may read a letter wrongly, causing it to fail in a limited amount of cases. In order to fix that and similar issues like plurals of words, this method goes letter by letter and returns a percentage with how similar to the original text a label is, doing this for each phrase inside that label



(a) Settings menu screenshot of FIFA 23 (b) Gamemodes on FIFA 23

Figure 4.10: Example of screenshots of different menus from FIFA 23

However, even with the subsequent improvements, there is a problem that these text classifiers cannot avoid, which is lack of context. For example, on Figure 4.10, on (a) the word "MATCH" can be interpreted by trained models as "play a match", when right here its context suggests it actually means "match settings". In addition to that, models can not detect proper nouns. Taking as example the word "SEASONS" on (b), models would assign a "season pass" label in the best of the scenarios, instead of "game mode". It does not matter how extensive the training is, it can not detect words it has not seen before.

Figure 4.11 is an example of an entry in the data returned by the tool after the text classification using the dictionary solution. As can be seen, the text *SOLO* is classified as *single player*. This completes the most vital information about the objects detected in the image.

```
{
    "Type": "Text",
    "Text": "SOLO",
    "Box": {
        "Left": 987,
        "Top": 319
        "Width": 125,
        "Height": 74,
    },
    "ContourBox": {
        "Left": 967,
        "Top": 299,
        "Width": 358,
        "Height": 299
    },
    "Labels": [
        {
            "Label": "single_player",
            "Confidence": 1.0
        }
    ]
}
```


Figure 4.11: Combined dictionary result example. This example shows the type of element, the text it contains with its position and dimensions as well as the position and dimensions of its contour, and finally its labels.

4.2.4. Fourth Iteration - Checkers and Highlighted Button

An addition to the tool is the use of checkers. Checkers, as we define them, are classes that check whether certain labels are contained in aligned groups of elements in order to find navigable menus. This is useful to detect components such as menu bars, or to detect which is the first menu found that fulfills the rules to be a main menu.

Thanks to the checkers, the context of the tool is updated, allowing it to understand better its current state. For instance, a common element in video game interfaces are menu bars. Knowing what a menu bar is, they will be considered as such, facilitating navigation.

In combination with checkers, there is a second improvement related to understanding the context of a video game state. Highlighted buttons are considered as the most differentiated buttons from a group. Two methods have been tested, being the first one detecting the highest mean deviation of V components in images with HSV format, and the second one doing histogram intersections in order to find the lowest intersection value. Although the first method works in some F1 2022 test cases, it does not always work, since the ratio between the number of background and text pixels is not always the same. For this reason, a generic comparison of histograms by applying intersections works better. Both of these additions are explained on subsubsection 3.3.3.3.

The basic process is the opposite: instead of looking for the most different button, the

tool focuses on what makes all these buttons similar and then excludes the one with the least similarity. This solution works on all tested games without any false detection (see chapter 5).

At the end of this whole Cogito navigation pipeline process the information is stored in a dictionary object and used to perform input. The information is stored as follows:

- `Data`: A list containing all the detected texts and icons stored in raw form. Each element maintains the same structure as Figure 4.11.
- `RawText`: A list which contains all the texts without further information. This list is used in order to determine if a menu has changed considerably from one input to another in the navigation.
- `MenuInfo`: This entry on the dictionary stores the different horizontal, vertical or grid menus that were determined by the checkers. Each menu contains a list of its elements and the index of the selected element at the moment.

4.2.5. Fifth Iteration - Lambda

The tool at this point consists of multiple ways to analyze what is on screen. In the future, another tool designed to navigate using the information currently obtained should be running at the same time. For this reason, and also for convenience, the objective of this iteration is porting the code so it can run on a python lambda, in order to facilitate remote access on another device that will be doing the navigation while communicating with this lambda.

Specifically, the service used to host this lambda is AWS Lambda, described on subsection 4.1.4. A continuous integration system updates the code with the newest updates, and uploads it to the lambda. Some basic tests can be done this way to guarantee that the tool is working as intended, but for now there are not any. In the future it would be a good idea to have them to speed up the work between iterations.

4.3. Architecture

Cogito is a console application implemented in Python and composed of two Python modules:

- **CogitoApp**, a console application that contains the main loop of the tool. This application is in charge of executing a given game, taking screenshots and sending them to the lambda function and based on the results of the function it will perform input to fulfill its objective. This application uses an *appsettings.json* file to define which game executable to use and an input module to navigate through a menu. The main focus of this project has been the Cogito module and this CogitoApp only contains what is necessary to execute a navigation loop.
- **Cogito**, a module stored as a lambda function in the AWS Lambda service. This module is the central part of the tool, since it is in charge of all image processing, element detection and classification process explained before (section 3.2). The lambda

function takes an image as input and returns a dictionary object containing all the detected and classified elements, as well as the type of menu and selected button. This module can also be used locally as a Python module.

The architecture of the Cogito module is divided into five sub-modules, each one in charge of a part of the image processing, element detection and classification process explained above (chapter 3). The following UML diagram graphically depicts the Cogito architecture (Figure 4.12).

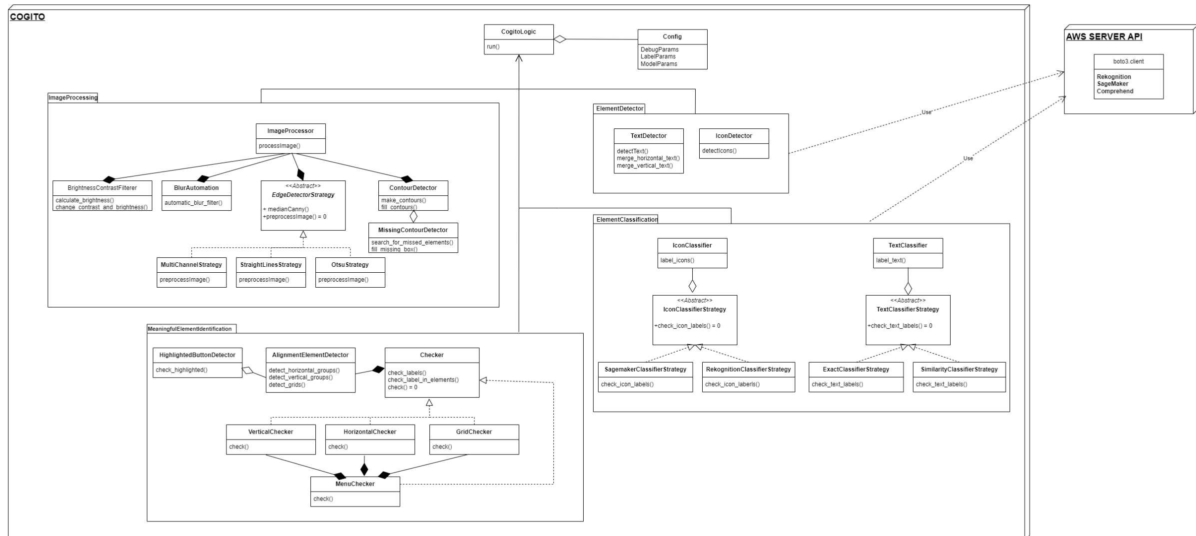


Figure 4.12: Cogito UML architecture diagram

4.3.1. Image Processing module

This module is divided into several classes that are then combined to obtain the result of contour and edge detection, the classes are as follows:

- `BrightnessContrastFilterer` - As the name suggests, this class is responsible for adjusting the brightness and contrast of an input image to better adapt to contour and edge detection.
- `BlurAutomation` - It uses the output of the `BrightnessContrastFilterer` class to apply the different types of blur filter by obtaining the sharpness and contrast as explained in subsection 3.2.2.
- `ContourDetector` - This class takes an image as input to detect its contours and contains methods to filter those contours based on text and icons.
- `MissingContourDetector` - As explained earlier in this section, this class detects missing contours that might not have been correctly recognized by the previous algorithms. It detects these contours based on alignment, distances and already existing contours.
- `ImageProcessor` - The `ImageProcessor` class is the main focus of this module, containing the rest of the classes to combine their results and return the final output.

- `EdgeDetectionStrategy` - An abstract class whose children define the different Canny algorithms to use. The different strategies are as follows:
  - `MultiChannelStrategy` - Applies the median Canny algorithm to each channel of the input image and combines them all.
  - `OtsuStrategy` - Applies the Otsu thresholding algorithm which returns an intensity threshold to define the background and foreground of an image (Sezgin and Sankur (2004)). This threshold is used for the Canny algorithm.
  - `StraightLineStrategy` - Uses OpenCv's straight line detection algorithm to detect only the horizontal and vertical lines of an image in combination with the Canny algorithm.

  The strategy used at the moment is the `MultiChannelStrategy`. This class is used by the `ImageProcessor` to properly detect the edges in a screenshot.

4.3.2. Element Detection module

The Element Detection module contains the two classes of machine learning used to detect features in an input image:

- `TextDetect` - This class is responsible for making calls to AWS Rekognition to detect text in an image. It has methods that based on the words detected by the model discards words not aligned horizontally and combines those words into sentences or paragraphs based on alignment, distance and font size.
- `IconDetect` - Similar to the `TextDetect`, is responsible for making calls to AWS Rekognition to detect icons in an image, this class needs a properly trained model to work with and currently uses a model trained and tested with over 250 images for general icon detection.

4.3.3. Element Classification module

It follows a similar structure containing two classes to classify text and icons respectively. They rely on Strategy classes to establish different types of classification.

- `TextClassifier and TextClassifierStrategy` - These classes are responsible for classifying the text detected by the Element Detection module. The TextClassifier class can take the output of the `TextDetect` class and classify it directly, it uses a label dictionary where each label contains a list of words related to it.

  This class uses the abstract class `TextClassifierStrategy` to define what kind of classification to use, at the moment, a `SimilarityClassifierStrategy` is used to compare the detected words with those in the label dictionary based on word similarity, but it can also be configured to use an `ExactClassifierStrategy`, which only labels detected words if they are exactly in the label dictionary.
- `IconClassifier and IconClassifierStrategy` - These classes follow the same structure as the text classification classes, but instead of using a dictionary of labels.

The `IconClassifier` uses an `IconClassifierStrategy` abstract class which calls a model that looks for common icons in a video game interface and classifies them. The Strategy used can implement a Rekognition or SageMaker model, which can be configured on the `models.py` script. Then the image is sent to the specified model, and each icon is classified with its label.

4.3.4. Meaningful Element Identification module

This module uses the classification results to find significant elements of a video game menu. It contains the following classes:

- `AlignmentElementDetector` - One of the pillars of the module, this class uses the results obtained in the detection of texts and icons to group these elements in different horizontal and vertical groups, according to their alignment and distance. This class stores the groups detected for the other classes to use.

- `Checker` - The Checker is an abstract class that takes an array of labels, an alignment detector and a highlighted button detector and uses them to check for significant items in a menu. Its way of checking is to look for whether a given number of labels from the label array are found in the groups of aligned items. It also defines functions to check specific labels in the data output returned by the classification and a simple set of rules for checking these significant items.

- `Horizontal, Vertical and Grid Checkers` - These classes inherit from the Checker class, they are in charge of checking for each type of group of elements if there are some labels of a given list of labels. This solution is used to determine if an aligned group of elements is truly part of a navigable menu, since elements in a menu should be related to specific menu labels.

- `HighlightedButtonDetector` - This is one of the most important classes of the tool, because knowing the exact button where the tool is located greatly aids navigation. The `HighlightedButtonDetector` uses a group of aligned elements and the original image to work properly. This class acquires the histogram of each element in the group and compares it using an intersection algorithm.

  Once all the intersections of the histogram of all the buttons are obtained, the sum of all the intersections of each button is done and the lower one will be the selected button, since if its intersections with the rest of the buttons is low it means that it does not look like the rest of the buttons which makes it the highlighted button.

  OpenCV allows the developer to define the size of the histogram to be more or less detailed. In the case of this class small size histograms are used due to the level of detail in some buttons which can complicate the detection, having a small histogram will keep the general details of the button and will allow a more accurate comparison between the other buttons.

4.3.5. Configuration module

In order to properly configure the tool a module was created. This configuration module contains two *.json* files which one stores the tool configuration (*appsettings.json*) and the

other stores the label dictionary (*labels.json*). Figure 4.13 shows an example of the label dictionary and Figure 4.14 an example of the tool configuration.

```
"achievements": [
    "quests",
    "trophies",
    "achievements",
    "trophy",
    "achievement",
    "awards"
],
"audio": [
    "audio",
    "sound",
    "audio settings",
    "volume control",
    "audio & rumble",
    "sound settings"
]
```

Figure 4.13: *Labels.json* example containing two labels.

This module contains the `CogitoConfiguration` class. This class stores the data read from both *.json* files for the rest of the tool to use. The `CogitoConfiguration` is in charge of storing the necessary parameters to configure the tool, which are:

- `DebugParams` - Contains options that allow the developer to choose which image processing steps to view. These options are represented as boolean values in the *appsettings.json* file.
- `LabelParams` - Stores the label dictionary, where each key in the *labels.json* file means a label and the value of each key is an array of words.
- `ModelParams` - Allows the developer to choose the different machine learning models easily, the information is contained in the *appsettings.json* file as strings of the different URLs for the models.

4.4. Conclusion

This chapter explains the development process and architecture of the Cogito tool. This process started with a research and prototyping phase in which existing tools were tested and their capabilities were recognized for the development of the tool.

Once this prototyping phase ended, the development began with an iterative process of constant improvement of the tool. These iterations marked important steps, focusing on core functionalities needed on each one of them, being integrated into the workflow of the tool and forming a proof of concept.

```
{
    "game_executable": "F1_23.exe",
    "local": true,
    "debug":  {
        "enabled": false,
        "debug_folders": {
            "output": true,
            "json": true,
            "blur": true,
            "canny": true,
            "contour": true,
            "contrast": true,
            "text_classification": true,
            "text_detection": true,
        }
    }
}
```

Figure 4.14: *Appsettings.json* example. It contains the game to be tested, whether lambda will be used for processing or locally and whether debug images will be generated.

The experience obtained researching about how Sikuli used the library OpenCV applying visual testing techniques made the process of investigating its functionalities easier. Although the image processing iteration was the longest one, investigating other tools was very helpful to get an idea of what could be done. Although other tool's ideas were not directly used, they have been studied as they may be useful on the future on navigation development.

In the next chapter the performance of the tool will be discussed, using different metrics.

Chapter 5

Evaluation

This chapter will discuss the performance of the current state of the tool, proposing two evaluations in order to understand the quality of the tool and also to have a good measure of which parts are more time consuming. Both will be focused on the same parts of the tool to determine if there are any that need improvement. This will be done from two points of view, quality evaluation and time performance evaluation.

5.1. Quality performance

The target of this evaluation will be to gauge the accomplishment of the tool, measuring how it performs in every important aspect of its workflow: Contour detection, text detection, icon detection, text classification, icon classification and highlighted tiles detection.

5.1.1. Methodology

Some aspects of this tool are based on machine learning models, being those the icon detector and the icon classification models. These had been trained as explained on section subsubsection 4.2.3.2, and for clarification, the dataset used in their training did not include images from the game is tested on, so the results will not be biased. For this evaluation, the tool will run throughout test cases from the video game F1® 2022, being those screenshots from all the menus from the game.

The way this is evaluated consists of running the tool throughout these screenshots and then check manually if the results match with the expected output, using thresholds. The reason to do it this way is to have a more accurate comparison between the real and the predicted results, since in some cases the predicted result could potentially be very different to the real one. This makes complicated to match automatically these elements, and while it could be done, it was determined that a manual comparison would be a better solution.

The acceptance criteria for each step is:

1. For contour detection, positions and sizes of created tiled buttons will be compared to the real ones, using a threshold to determine if a contour is correct or not. Indications of failure are: multiple contours wrongly mixing into only one, or a contour separating into multiple ones.

2. For text detection, texts detected by the OCR will be compared to real ones, and to determine if they are correct the characters that are numbers or letters need to be exactly similar, excluding the rest of them from this evaluation. Indications of failure are: letters and numbers are detected wrongly.

3. For icon detection, predicted icons positions will be compared to real ones, using a threshold to indicate if they are correct or not. Only icons that appear fully count for this so cropped icons are not considered. Indications of failure are: a icon is detected where there is nothing, or an icon is not labeled as such.

4. For text classification, the result provided by the dictionaries will be compared to the expected one, which needs to be the same one to be correct. Indications of failure are: text is labeled as "undefined", or text is labeled wrongly.

5. For icon classification, predicted icon labels will be compared to the expected ones, which needs to be the same one to be correct. Indications of failure are: icon are labeled as "undefined", or icon are labeled incorrectly.

6. For the highlighted tiles detector, the predicted selected buttons will be compared to the real selected buttons, and both need to have the same button selected to be correct. Indications of failure are: a predicted selected button is not a real selected button, or a group with a selected element does not label any tile buttons as selected.

In addition to this, these comparisons will be done element by element of every image, counting the number of successes and failures. Elements that are shared between screenshots will not be taken into account, and they are counted only once.

5.1.2. Results and conclusions

This evaluation has been done throughout the 129 images that contain every menu for F1® 2022. In order to show better the confidence of the predictions done by the model, confusion matrices are used. A confusion matrix indicates the number of false positives, false negatives, true positives and true negatives of predictions made by machine learning models. This has been used to gather all the information that has been obtained using the already explained acceptance criteria.

It should be clarified that even when not all parts are machine learning (f.e the dictionaries for text classification), it has still been useful since all these parts provide predictions that can be compared to a real result. This has caused not all metrics to have true negatives, since it does not make sense in some cases, as for example with contour detection. A contour can be detected correctly (true positive), wrongly detected (false positive). It also can not exist (true negative), but it does not make sense to ask for contours not detected that did not exist in the first place. These fields have been marked as 0.

As Figure 5.1 shows, the quality of the tool is measured on each step, by comparing the number of predicted against the number of expected values. Also the precision and

Contour detector			
		Predicted	
		Positive	Negative
Actual	Positive	501	53
	Negative	15	0
Precision		0,904	
Recall		1,000	

Text detector			
		Predicted	
		Positive	Negative
Actual	Positive	378	8
	Negative	0	0
Precision		0,979	
Recall		1,000	

Icon detector			
		Predicted	
		Positive	Negative
Actual	Positive	73	11
	Negative	32	0
Precision		0,869	
Recall		1,000	

Text classification			
		Predicted	
		Positive	Negative
Actual	Positive	52	75
	Negative	9	0
Precision		0,409	
Recall		1,000	

Icon classification			
		Predicted	
		Positive	Negative
Actual	Positive	15	12
	Negative	7	0
Precision		0,556	
Recall		1,000	

Highlighted buttons			
		Predicted	
		Positive	Negative
Actual	Positive	26	58
	Negative	24	12
Precision		0,310	
Recall		0,684	

Figure 5.1: Confusion matrix for different steps of the tool through F1® 2022 screenshots

recall for each step is displayed, although it only makes sense to measure recall on the highlighted button detection step, where there can be true negatives.

The results of this evaluation show clearly in which steps the tool needs more work to be done. While the results obtained for contour and text detection are reliable with minimum wrong predictions, the ones obtained by the highlighted button detector are not very trustworthy.

In addition to this, text classification is not performing as well as expected. This is due to the dictionary being made of the most common words for each category, but it has not been adjusted to every game mode from this game. This causes the results to be worse than it could be, but shows that more entries should be added in order to detect more words correctly. As it was explained on subsection 3.3.1, game modes have not been able to be detected even by machine learning models, because they could potentially mean different things outside an specific video game.

Next, icon detection has performed reasonably good. Although the number of false positives is a little concerning, a wrongly detected icon is not a critical error for the tool. Still, more work should be done to tune this model into a better version that avoids background elements in 3d perspectives, only sticking to the 2d icons that appear on menus.

Finally, icon classification shows its limited utility. Comparing the precision between this step and icon detection step shows that this model performs worse. That makes sense, since dataset it was trained with is smaller and filtered between all its labels, as it was explained on subsubsection 4.2.3.2. However, increasing the amount of images per label on training, the precision should improve, to the point of being reliable for very generic icons such as a settings icon.

Even so, these results show that the level of quality in each section provide a good start for a proof of concept. It can be seen that there is considerable room for improvement, since it is still far from being a definitive and closed tool with results that can be trustworthy.

5.2. Time performance

As a matter of fact, obtaining better times has not been the primary objective of this proof of concept, but is still necessary to understand the current state of the tool. The main goal of this evaluation will finding the bottlenecks on the workflow of the tool in order to know which parts need to be more time efficient.

5.2.1. Methodology

This evaluation has been done monitoring the time employed by each step on every image from the last evaluation. In addition to the images from F1® 2022, time has also been monitored on screenshots from another video game, FIFA 2023. This time measurement has only been done once due to the amount of images used, which should be descriptive enough about the state of the tool.

Time has been measured on the following steps: contrast and brightness, blur filter, edge detector, contour detector, text detector, icon detector, text classification, icon classification and highlighted tiles detection.

It is important to note that these services include network latency on the calls to the models responsible of text detection, icon detection, and icon classification, which are cloud-based online services, but the rest of the steps are done locally on a virtual machine with 16 CPU, 110 GB of RAM, and an NVIDIA Tesla T4 as GPU (1 GPU and 16 GB).

The metrics used to determine how much time it takes for the tool to process an image will the calculated this way:

1. Time events will be stored on a list each time an important part of the tool starts and finishes. This is repeated for each image and it is done in the same way a telemetry system would work, but at a much smaller scale.
2. When these two time stamps are obtained, a subtraction will be done. The time obtained will be calculated by subtracting the end time from the start time, and then final times will be stored on a data frame (for easier later display of the results)

With this information, it can also be calculated the total time needed to process each image.

5.2.2. Results and conclusions

The same number of images has been used than in the previous evaluation: 129 for F1® 2022, and 72 for FIFA 2023[1]. The full results are included on a drive folder, but only a fraction of those are included on the tables from the figures since not all entries are needed to understand the objective.

As it can be seen on Figure 5.2 and Figure 5.3, in both tables each column show the time needed for each step for each image of each game. All results are sorted by the total time

[1] Full results area available at https://drive.google.com/drive/folders/1on6ZYTw0sAniN4AEGEaUqITnbzMLw-dC?usp=sharing

needed to process an image, going from the lower to the highest times. The slowest step is highlighted in red, and the total processing time for that image is highlighted in green. Analyzing the results from both games the same limitation appears. One of the most time consuming parts is detecting the text on the screen using Rekognition's OCR model. In addition to this, there is another bottleneck that appears mostly on FIFA 2023 sometimes when adding blur filters, which can be seen on the third column. More specifically, it has been observed that higher times only occur when Bilateral filters are applied, although this information is not visible on the table. The reason why this is chosen automatically for the most fitting images and why the filter takes longer is explained on subsection 3.2.2. This is due to it is based on a time consuming algorithm which takes each pixel and converts it to the weighted average of its neighbours, and the time needed really depends on the hardware on which the tool is running.

An obvious limitation this proof of concept has is working on a single thread. The use of synchronous calls to the models used on text detection, icon detection, and icon classification forces the tool to wait until models have done their work. However, there are parts of the tool that could be working at the same time. If calls to these three models were done at the same time while blur filters are applied, a lot of time could be potentially saved.

On Figure 5.4 and Figure 5.5 it can be seen the results from the tables in the shape of graphs. Observing the metrics this way shows in a more obvious way how time is wasted waiting for the response of machine learning models, when it is not really necessary to wait for their results since the content of these models is not needed to calculate previous steps.

Finally, on Figure 5.7 and Figure 5.6 boxplot diagrams can be seen. These are only to prove that the steps of blue filter and text detection keep the values shown on the time tables. This can be seen on the blue lines inside the boxes, representing the median of the seconds per step of each image.

5.3. Conclusions

After obtaining these results, some points have been clarified. First, the quality of machine learning models related to icons needs to be improved. While the icon detection model has shown results that start to get close to being reliable, it still is not, and neither the icon classification is. More training should be done with a better dataset to increase their confidence.

In addition to this, other steps not related to machine learning need more work, which is the case of the highlighted buttons detector. Now it has to be studied the reason why these results aren't reliable. The high amount of false negatives in comparison with false positives included on Figure 5.1 seem to indicate something is failing in on some images where no highlighted button is detected at all, so that should be studied.

Finally, the total times obtained for this proof of concept are reasonable. However, the whole process of the tool is expected to run multiple times on tests. For this reason, time should be reduced as much as possible to avoid unreasonable long tests in the future.

An effective way to eliminate the problem of wasting execution time would be to use a multi-threaded implementation, as it would reduce the unproductive waits to a minimum, achieving a possible improvement of reducing the time needed on F1® by half in the best

of situations.

	image_name	contrast_and_brightness	blur_filter	edge_detector	contour_detector	text_detector	icon_detector	text_classification	icon_classification	highlighted_buttons	total_time
0	fifa23 (49).png	0,009	0,036	0,157	0,010	1,839	1,038	0,003	0,755	0,000	3,847
1	fifa23 (36).png	0,009	0,036	0,156	0,097	2,299	1,134	0,019	0,800	0,001	4,551
2	fifa23 (43).png	0,009	0,036	0,147	0,107	2,182	1,567	0,010	0,939	0,001	4,997
3	fifa23 (42).png	0,009	0,034	0,148	0,145	2,571	1,550	0,022	0,939	0,001	5,420
4	fifa23 (38).png	0,009	0,035	0,153	0,259	2,719	1,592	0,028	0,952	0,003	5,750
5	fifa23 (59).png	0,009	0,034	0,143	0,269	2,652	1,742	0,030	0,966	0,003	5,848
6	fifa23 (58).png	0,010	0,036	0,157	0,344	2,749	1,639	0,038	0,944	0,004	5,921
7	fifa23 (5).png	0,009	0,035	0,152	0,164	2,620	1,876	0,022	1,061	0,002	5,941
8	fifa23 (2).png	0,008	0,034	0,164	0,363	2,613	1,819	0,022	1,067	0,002	6,092
9	fifa23 (16).png	0,008	0,035	0,158	0,129	2,729	1,939	0,021	1,093	0,002	6,114
10	fifa23 (40).png	0,010	0,038	0,156	0,189	2,976	1,852	0,022	1,093	0,002	6,338
11	fifa23 (39).png	0,009	0,035	0,166	0,604	2,691	1,901	0,026	1,096	0,003	6,531
12	fifa23 (60).png	0,010	0,035	0,152	0,271	3,444	1,638	0,029	1,084	0,003	6,666
13	fifa23 (37).png	0,009	0,035	0,163	0,443	3,181	1,826	0,037	1,019	0,003	6,716
14	fifa23 (1).png	0,009	0,036	0,157	0,347	3,127	1,920	0,028	1,103	0,004	6,731
15	fifa23 (44).png	0,008	0,035	0,150	0,084	3,667	1,953	0,021	1,072	0,001	6,992
16	fifa23 (50).png	0,008	0,036	0,154	0,029	4,367	1,640	0,004	0,938	0,000	7,176
17	fifa23 (41).png	0,009	0,035	0,134	0,236	4,171	1,591	0,023	1,021	0,002	7,222
18	fifa23 (9).png	0,009	2,955	0,150	0,061	2,332	1,073	0,028	0,809	0,003	7,420
19	fifa23 (71).png	0,009	2,766	0,144	0,080	2,100	1,414	0,017	0,894	0,002	7,427
20	fifa23 (4).png	0,009	0,034	0,180	0,525	3,683	1,936	0,023	1,093	0,003	7,487
21	fifa23 (53).png	0,009	0,035	0,157	0,518	4,151	1,731	0,041	0,979	0,005	7,626
22	fifa23 (12).png	0,009	2,694	0,161	0,057	2,106	1,749	0,016	0,893	0,002	7,688
23	fifa23 (13).png	0,009	2,904	0,149	0,090	2,162	1,533	0,016	0,903	0,002	7,769
24	fifa23 (11).png	0,009	3,185	0,155	0,052	2,112	1,434	0,016	0,873	0,002	7,839
25	fifa23 (7).png	0,010	2,378	0,171	0,271	2,569	1,613	0,021	0,947	0,002	7,982
26	fifa23 (34).png	0,009	0,035	0,134	0,460	4,593	1,710	0,039	1,008	0,004	7,993
27	fifa23 (31).png	0,009	0,036	0,154	0,677	4,242	1,790	0,036	1,046	0,004	7,995
28	fifa23 (8).png	0,008	2,636	0,144	0,060	3,327	1,162	0,027	0,802	0,003	8,168
29	fifa23 (14).png	0,009	3,463	0,145	0,050	2,183	1,454	0,016	0,859	0,002	8,182
30	fifa23 (48).png	0,009	2,649	0,164	0,085	2,552	1,880	0,015	1,013	0,001	8,368
31	fifa23 (23).png	0,009	2,820	0,154	0,122	2,854	1,505	0,020	0,917	0,002	8,404
32	fifa23 (47).png	0,009	2,715	0,160	0,156	2,578	1,807	0,017	1,015	0,001	8,458
33	fifa23 (15).png	0,008	3,084	0,151	0,075	2,893	1,426	0,015	0,884	0,002	8,539
34	fifa23 (33).png	0,009	0,035	0,151	0,742	4,641	1,835	0,037	1,096	0,005	8,552
35	fifa23 (10).png	0,009	3,495	0,143	0,095	2,493	1,487	0,016	0,929	0,002	8,670
36	fifa23 (26).png	0,008	3,117	0,149	0,238	2,380	1,750	0,020	1,026	0,003	8,692
37	fifa23 (70).png	0,009	3,196	0,154	0,078	2,936	1,443	0,017	0,860	0,002	8,695
38	fifa23 (54).png	0,010	0,035	0,166	1,284	4,290	1,806	0,039	1,128	0,005	8,763
39	fifa23 (35).png	0,009	3,048	0,147	0,079	3,643	1,113	0,029	0,760	0,004	8,833
40	fifa23 (46).png	0,009	2,969	0,145	0,219	2,678	1,824	0,019	1,028	0,001	8,893
41	fifa23 (25).png	0,009	3,676	0,150	0,120	2,631	1,477	0,022	0,928	0,003	9,017
42	fifa23 (6).png	0,008	3,147	0,163	0,345	2,725	1,607	0,021	1,042	0,003	9,061
43	fifa23 (57).png	0,009	3,125	0,149	0,389	2,844	1,632	0,029	0,987	0,004	9,168
44	fifa23 (45).png	0,009	2,978	0,137	0,146	2,860	1,927	0,019	1,097	0,001	9,175
45	fifa23 (24).png	0,009	3,302	0,134	0,277	2,503	1,868	0,019	1,079	0,002	9,194
46	fifa23 (52).png	0,010	0,034	0,153	1,205	5,110	1,738	0,038	1,010	0,005	9,303
47	fifa23 (27).png	0,009	3,306	0,165	0,245	2,664	1,858	0,021	1,052	0,003	9,324
48	fifa23 (72).png	0,009	3,338	0,145	0,057	3,639	1,412	0,017	0,872	0,002	9,491
49	fifa23 (63).png	0,009	3,408	0,161	0,428	2,644	1,776	0,023	1,049	0,004	9,502
50	fifa23 (62).png	0,010	3,025	0,158	0,320	3,248	1,724	0,021	0,999	0,003	9,507
51	fifa23 (3).png	0,009	3,828	0,159	0,387	2,618	1,630	0,027	0,938	0,003	9,600
52	fifa23 (18).png	0,009	3,301	0,149	0,212	2,949	1,959	0,022	1,128	0,002	9,732
53	fifa23 (61).png	0,009	4,136	0,163	0,196	2,459	1,791	0,020	1,090	0,002	9,867
54	fifa23 (22).png	0,009	3,358	0,150	0,085	3,439	1,852	0,019	0,956	0,003	9,872
55	fifa23 (51).png	0,009	3,072	0,158	0,313	3,923	1,559	0,029	0,973	0,004	10,040
56	fifa23 (19).png	0,008	2,732	0,172	0,212	4,057	1,960	0,020	1,055	0,001	10,218
57	fifa23 (29).png	0,009	3,184	0,141	0,338	3,927	1,733	0,033	1,003	0,004	10,373
58	fifa23 (65).png	0,008	2,972	0,133	0,278	4,147	1,900	0,024	1,127	0,002	10,591
59	fifa23 (66).png	0,008	2,681	0,144	0,311	4,562	1,900	0,023	1,104	0,003	10,736
60	fifa23 (69).png	0,009	3,052	0,148	0,286	4,132	1,958	0,023	1,127	0,003	10,739
61	fifa23 (21).png	0,009	3,316	0,170	0,215	4,088	1,951	0,022	1,079	0,002	10,853
62	fifa23 (64).png	0,009	3,226	0,142	0,379	4,657	1,977	0,024	1,078	0,002	11,494
63	fifa23 (67).png	0,009	2,964	0,144	0,311	5,137	1,920	0,023	1,093	0,002	11,604
64	fifa23 (17).png	0,009	3,086	0,156	0,241	5,058	1,940	0,022	1,121	0,002	11,636
65	fifa23 (56).png	0,009	4,076	0,165	1,079	3,601	1,747	0,038	1,034	0,005	11,754
66	fifa23 (20).png	0,009	3,192	0,155	0,217	5,418	1,935	0,023	1,091	0,002	12,043
67	fifa23 (30).png	0,008	2,855	0,176	1,235	5,003	1,790	0,037	1,088	0,004	12,197
68	fifa23 (68).png	0,009	3,437	0,155	0,309	5,362	1,893	0,023	1,100	0,002	12,290
69	fifa23 (32).png	0,009	3,282	0,162	0,791	6,362	1,694	0,038	1,003	0,004	13,346
70	fifa23 (28).png	0,009	4,429	0,153	1,997	4,277	1,798	0,042	1,041	0,009	13,756
71	fifa23 (55).png	0,009	4,114	0,149	2,087	5,807	1,692	0,037	1,027	0,006	14,928

Figure 5.2: Table of times of each step of the tool applied to FIFA 2023 screenshots (in seconds)

	image_name	contrast_and_brightness	blur_filter	edge_detector	contour_detector	text_detector	icon_detector	text_classification	icon_classification	highlighted_buttons	total_time
0	f122 (56).png	0,009	0,036	0,142	0,007	1,828	0,857	0,007	0,705	0,000	3,591
1	f122 (53).png	0,009	0,037	0,139	0,005	2,432	0,894	0,003	0,715	0,000	4,234
2	f122 (90).png	0,002	0,010	0,041	0,004	2,797	1,143	0,011	0,771	0,001	4,780
3	f122 (115).png	0,009	0,037	0,165	0,007	2,044	1,640	0,008	0,909	0,000	4,820
4	f122 (117).png	0,010	0,037	0,144	0,006	2,136	1,627	0,008	0,898	0,001	4,867
5	f122 (45).png	0,008	0,035	0,152	0,014	2,351	1,797	0,019	1,003	0,002	5,381
6	f122 (14).png	0,009	0,036	0,142	0,017	2,281	1,809	0,018	1,070	0,002	5,384
7	f122 (17).png	0,010	0,038	0,144	0,011	2,373	1,780	0,018	1,060	0,002	5,436
8	f122 (61).png	0,009	0,035	0,147	0,011	2,200	1,953	0,010	1,104	0,000	5,469
9	f122 (38).png	0,008	0,034	0,154	0,028	2,452	1,782	0,026	1,003	0,002	5,490
10	f122 (126).png	0,009	0,038	0,169	0,018	2,200	2,004	0,006	1,048	0,001	5,494
11	f122 (16).png	0,010	0,033	0,159	0,011	2,597	1,763	0,018	0,992	0,002	5,585
12	f122 (116).png	0,010	0,037	0,173	0,006	2,850	1,605	0,007	0,908	0,001	5,597
13	f122 (39).png	0,009	0,035	0,158	0,033	2,495	1,768	0,025	1,079	0,002	5,604
14	f122 (46).png	0,009	0,035	0,158	0,011	2,293	1,985	0,009	1,106	0,000	5,606
15	f122 (125).png	0,009	0,036	0,174	0,020	2,262	2,016	0,006	1,086	0,000	5,609
16	f122 (37).png	0,009	0,035	0,148	0,027	2,623	1,739	0,026	1,056	0,002	5,665
17	f122 (36).png	0,009	0,034	0,163	0,029	2,581	1,797	0,027	1,042	0,003	5,685
18	f122 (65).png	0,009	0,034	0,137	0,010	2,313	2,064	0,006	1,148	0,001	5,722
19	f122 (34).png	0,009	0,034	0,174	0,063	2,623	1,799	0,028	1,027	0,002	5,759
20	f122 (44).png	0,008	0,034	0,149	0,013	2,852	1,698	0,019	1,045	0,002	5,820
21	f122 (41).png	0,009	0,035	0,157	0,013	2,957	1,743	0,021	0,987	0,001	5,924
22	f122 (35).png	0,008	0,035	0,149	0,030	2,926	1,731	0,026	1,021	0,002	5,928
23	f122 (48).png	0,010	0,036	0,152	0,010	2,567	2,075	0,009	1,084	0,000	5,944
24	f122 (47).png	0,009	0,035	0,155	0,006	2,479	2,107	0,009	1,147	0,000	5,947
25	f122 (64).png	0,009	0,034	0,150	0,009	2,638	2,097	0,007	1,067	0,000	6,011
26	f122 (33).png	0,009	0,035	0,154	0,035	2,937	1,755	0,028	1,068	0,003	6,025
27	f122 (18).png	0,010	0,034	0,160	0,012	2,993	1,774	0,017	1,046	0,002	6,048
28	f122 (51).png	0,009	0,032	0,171	0,011	2,329	2,307	0,006	1,187	0,001	6,053
29	f122 (19).png	0,009	0,034	0,148	0,015	2,965	1,853	0,018	1,013	0,002	6,057
30	f122 (42).png	0,009	0,036	0,150	0,011	3,159	1,669	0,020	1,005	0,001	6,060
31	f122 (78).png	0,008	0,034	0,141	0,010	2,576	2,241	0,012	1,092	0,000	6,114
32	f122 (88).png	0,008	0,034	0,155	0,012	2,376	2,289	0,011	1,231	0,001	6,117
33	f122 (81).png	0,009	0,036	0,148	0,012	2,494	2,271	0,012	1,180	0,001	6,163
34	f122 (75).png	0,009	0,034	0,146	0,012	2,595	2,232	0,013	1,128	0,000	6,169
35	f122 (102).png	0,009	0,035	0,149	0,009	2,536	2,290	0,012	1,130	0,001	6,170
36	f122 (79).png	0,009	0,034	0,148	0,012	2,576	2,213	0,012	1,167	0,000	6,171
37	f122 (74).png	0,009	0,035	0,150	0,009	2,606	2,217	0,013	1,152	0,001	6,192
38	f122 (82).png	0,009	0,034	0,164	0,011	2,565	2,268	0,013	1,128	0,001	6,193
39	f122 (50).png	0,010	0,033	0,142	0,008	2,662	2,234	0,008	1,098	0,000	6,195
40	f122 (85).png	0,009	0,034	0,159	0,009	2,574	2,285	0,013	1,133	0,001	6,217
41	f122 (6).png	0,009	0,034	0,161	0,026	2,557	2,282	0,018	1,155	0,002	6,244
42	f122 (107).png	0,009	0,035	0,152	0,011	2,647	2,243	0,013	1,142	0,001	6,253
43	f122 (5).png	0,009	0,034	0,156	0,038	2,581	2,271	0,018	1,151	0,002	6,260
44	f122 (93).png	0,008	0,034	0,147	0,010	2,568	2,214	0,013	1,266	0,000	6,260
45	f122 (87).png	0,009	0,034	0,147	0,011	2,690	2,252	0,013	1,115	0,001	6,272
46	f122 (8).png	0,009	0,036	0,157	0,032	2,618	2,227	0,021	1,170	0,003	6,274
47	f122 (112).png	0,008	0,035	0,154	0,021	2,684	2,178	0,013	1,203	0,001	6,297
48	f122 (96).png	0,008	0,035	0,151	0,009	2,550	2,323	0,015	1,209	0,001	6,301
49	f122 (108).png	0,009	0,035	0,154	0,010	2,756	2,201	0,013	1,125	0,001	6,304
50	f122 (105).png	0,009	0,034	0,164	0,007	2,654	2,289	0,013	1,137	0,001	6,307
51	f122 (106).png	0,009	0,036	0,155	0,009	2,629	2,246	0,011	1,215	0,001	6,311
52	f122 (60).png	0,009	0,035	0,146	0,012	2,896	2,097	0,011	1,108	0,000	6,314
53	f122 (10).png	0,009	0,035	0,149	0,027	2,645	2,200	0,020	1,233	0,002	6,320
54	f122 (68).png	0,008	0,034	0,159	0,010	2,736	2,234	0,015	1,125	0,001	6,322
55	f122 (43).png	0,009	0,034	0,155	0,012	3,384	1,701	0,019	1,015	0,002	6,331
56	f122 (52).png	0,008	0,034	0,151	0,013	2,665	2,290	0,005	1,168	0,000	6,334
57	f122 (71).png	0,008	0,036	0,140	0,010	2,772	2,221	0,013	1,140	0,001	6,341
58	f122 (3).png	0,009	0,036	0,144	0,052	2,667	2,257	0,018	1,167	0,002	6,351
59	f122 (99).png	0,009	0,034	0,133	0,010	2,577	2,293	0,012	1,298	0,001	6,367
60	f122 (98).png	0,009	0,036	0,148	0,013	2,587	2,309	0,016	1,251	0,001	6,370
61	f122 (94).png	0,008	0,035	0,157	0,009	2,776	2,238	0,014	1,149	0,001	6,387
62	f122 (84).png	0,009	0,035	0,155	0,012	2,751	2,250	0,012	1,177	0,001	6,402
63	f122 (69).png	0,009	0,034	0,142	0,013	2,787	2,250	0,016	1,151	0,001	6,403
64	f122 (95).png	0,009	0,034	0,166	0,012	2,776	2,258	0,016	1,158	0,001	6,430
65	f122 (4).png	0,009	0,035	0,159	0,026	2,807	2,287	0,019	1,103	0,002	6,448
66	f122 (12).png	0,009	0,035	0,163	0,019	2,843	2,242	0,020	1,131	0,002	6,464
67	f122 (49).png	0,008	0,034	0,166	0,011	3,003	2,156	0,008	1,081	0,000	6,466
68	f122 (83).png	0,009	0,035	0,169	0,010	2,855	2,258	0,013	1,117	0,001	6,467
69	f122 (76).png	0,009	0,034	0,147	0,010	2,897	2,235	0,012	1,130	0,001	6,475
70	f122 (111).png	0,009	0,035	0,152	0,018	2,902	2,218	0,013	1,143	0,001	6,491
71	f122 (7).png	0,009	0,034	0,154	0,027	2,543	2,548	0,021	1,164	0,002	6,502

Figure 5.3: Table of times of each step of the tool applied to F1® 2022 screenshots (in seconds)

Figure 5.4: Graph of times of each step of the tool applied to FIFA 2023 screenshots (in seconds)

Figure 5.5: Graph of times of each step of the tool applied to F1® 2022 screenshots (in seconds)

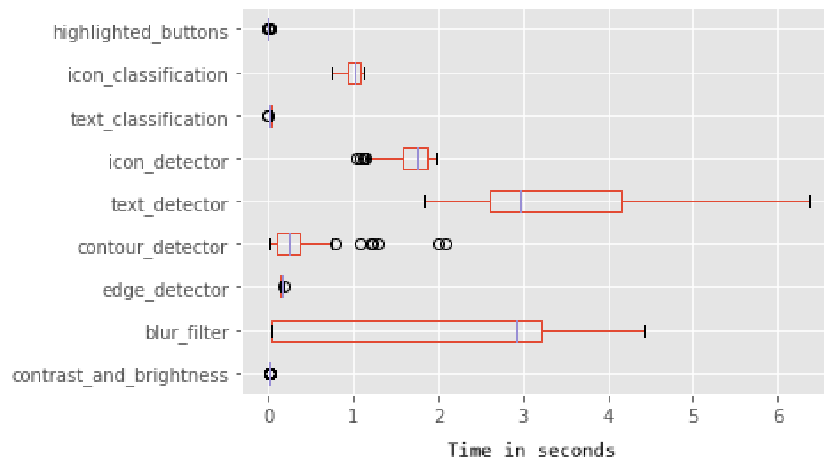


Figure 5.6: Boxplot of times of each step of the tool applied to FIFA 2023 screenshots (in seconds)

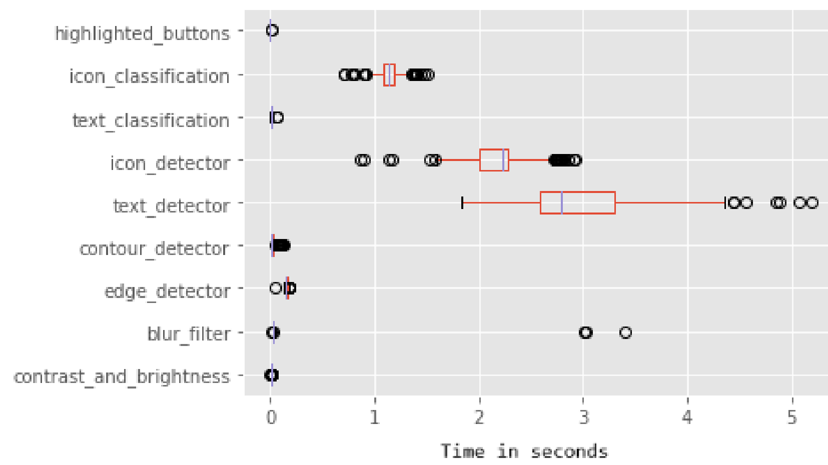


Figure 5.7: Boxplot of times of each step of the tool applied to F1® 2022 screenshots (in seconds)

Chapter 6

Conclusions and Future Work

Manual testing is a costly and tedious job that involves a large number of testers performing simple tasks that are performed repetitively over a large number of video games. The job of these testers should be to focus on testing those elements of the game that are complex and thus have a more polished game. This is why an automatic navigation tool would be ideal for testing a video game, since the menus in the industry are usually similar and testing in such menus is relatively simple.

This project has described the design and development of the Cogito tool, which through the use of image processing, element detection and classification is able to identify the elements of a video game menu using a screenshot and perform the necessary input to navigate through that menu. Cogito is divided into two parts: (1) writing a script based on human-like commands such as *Go to Settings* and (2) visual navigation to run the script. This navigation is mainly focused on detecting outlines of buttons, text and icons, with that information it is able to classify the elements and performs input based on it. This project has been mainly focused on the navigation part of the tool.

The development of Cogito started with a prototyping phase in which team members looked for capabilities to implement from other existing tools. Once all the necessary information was gathered, the iterative development process began, with each iteration resulting in an improved version of the tool. This iterative process was strongly based on the SCRUM agile methodology.

Cogito started as a proof of concept that over time grew into a project with a lot of potential, which can be worked on further to correctly navigate through a menu as a scripting language for testers to use.

Now, Cogito is a tool in development which has been implemented as described throughout this document, implemented on python. It makes use of a potent library for image processing, OpenCV, as well as implements machine learning models using Amazon Web Services for the specific tasks of text detection, icon detection and icon classification.

An evaluation of the tool has been done, and it has been determined that it still has lots of room for improvement referring both to quality and times of each step. However, it is clear now which steps need to be more efficient, and which steps need to improve their predictions. After working on the issues mentioned, the tool could significantly improve.

The following section will discuss the improvements that can be applied to Cogito and future work for the tool.

6.1. Future work

Although the results obtained are promising, the tool is not perfect. Since it is a proof of concept, there is plenty of room for improvement. A few items about the most important upgrades that can be done would be:

- Continue developing a separated navigation app that relies on Cogito implementation to understand and store states of the game. This app would run locally where the game is running, and using information provided by the Cogito running on AWS Lambda, it would understand in what menu the game is, which ones are the possible input actions, and how to find a specific text by using labels.

  An additional use for the tool that does not rely on labeling would be a menu crawler. After obtaining the structure of a menu using visual testing techniques, functional testing ones could be applied over it in order to create a graph that maps each different node from the graph to a specific menu. This would be useful to automatically find out if all the menus before launching a game mode are accessible, which is an important common task done while testing GUIs.

- Doing more research about possible ways to reduce unproductive waiting times. One way would include a multi thread implementation, as it was mentioned on the time performance conclusions (subsection 5.2.2), that calls the models in parallel while executing the image processing steps on another thread, and then only waiting when it actually reaches the point they are needed. That would allow minimizing the total time for the process, reducing the unproductive waits to zero if the models finish their tasks before reaching the point they are needed.

- Improve efficiency in some of the most time consuming steps of image processing, such as applying bilateral filters, which can take whole seconds to be applied. A possible way that has not been tested on this project to improve this is using Fourier Transform, a process that is based on the idea of all functions can be approximated as the sum of infinite sinus and cosines. Image operations may be optimized using these transformations.

- Once the datasets used by Rekognition models have shown good results, these could be converted to the format SageMaker uses. This can be done with a script that was created during the creation of the first SageMaker models, which takes a .lst file and converts it to a .manifest one.

  The reason to do this is SageMaker allows to have more control of model training. It does not necessarily mean the results are going to be better, since this has been already tried with the general icons model obtaining a worse mAP. Even so, it would be worth trying, but it is not a critical issue right now.

- Train AWS Comprehend models to be used in combination with the already existing checkers in order to get a better context of the situation of interfaces in video games. This could be used for very specific menus, such as settings, popups, or "press x button to play" menus. These would be good examples of use since the text contained

in these menus is always very similar, allowing a possible integration of Comprehend within the tool that runs a custom classifier model for these menus.

However, it would only work with detailed menus. More abstract menus such as a main menu, which can contain words related to all the menus mentioned before, would not work on this model. Checkers still will have a reason to exist in these cases, which is why a combination of both would give better results.

Bibliography

Cambridge dictionary. 2023. Avail. at https://dictionary.cambridge.org (last access, Aug, 2023).

Electronic Arts. Apex Legends. 2019. Avail. at https://www.ea.com/en-gb/games/apex-legends (last access, Sep, 2023).

Baresi, L. and Pezzè, M. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, Vol. 148(1), 89–111, 2006. ISSN 1571-0661. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

Bradski, G. and Kaehler, A. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, 2008.

Chang, T.-H. and Yeh, T. Sikuli. 2009. Avail. at http://sikulix.com/ (last access, April, 2023).

Cuellar, D. Appium. 2012. Avail. at https://appium.io/ (last access, Sep, 2023).

Delapouite and Lorc. Video game icons dataset. 2022. Avail. at https://game-icons.net/ (last access, Sep, 2023).

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. 2019. Avail. at https://arxiv.org/pdf/1810.04805.pdf (last access, Sep, 2023).

Gandhi, T. K., Konar, D., De, S., Dey, S. and Bhattacharyya, S. *Advanced Machine Vision Paradigms for Medical Image Analysis*. Elsevier Science, 2020.

Gomez, J. and Olfman, L. The impact of combining gestalt theories with interface design guidelines in designing user interfaces. *18th Americas Conference on Information Systems 2012, AMCIS 2012*, Vol. 3, 2367–2374, 2012. Avail. at https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1436&context=amcis2012 (last access, Sep, 2023).

Gonzalez, R. and Woods, R. *Digital Image Processing*. Financial Times Prentice Hall, 2017.

Gutierrez, R., Evans, S. and Sephew, P. Gamedriver. 2018. Avail. at https://gamedriver.io/ (last access, April, 2023).

HUGGINS, J. Selenium. 2004. Avail. at https://www.selenium.dev/about/ (last access, April, 2023).

JAIN, R., KASTURI, R. and SCHUNCK, B. G. *Machine Vision*. McGraw-Hill Inc, 1995.

KEITH, C. *Agile game development with Scrum*. Addison-Wesley, 2015.

MATHWORKS. Machine learning vs deep learning. 2017. Avail. at https://uk.mathworks.com/videos/introduction-to-deep-learning-machine-learning-vs-deep-learning-1489503513018.html?s_tid=vid_pers_recs (last access, Sep, 2023).

SEZGIN, M. and SANKUR, B. Survey over image thresholding techniques and quantitative performance evaluation. *J. Electronic Imaging*, Vol. 13(1), 146–168, 2004.

SMITH, R. and HEWLETT-PACKARD. Tesseract. 1984. Avail. at https://github.com/tesseract-ocr/tesseract (last access, Sep, 2023).

SMITH, R. W. *The extraction and recognition of text from multimedia document*. PhD thesis, The University of Bristol, 1987.

SUTHERLAND, J. *Scrum: The Art of Doing Twice the Work in Half the Time*. Random House Business, 2015.

SZELISKI, R. *Computer Vision: Algorithms and Applications*. Springer Nature, 2010.

TODOROVIC, D. Gestalt principles. 2008. Avail. at http://www.scholarpedia.org/article/Gestalt_principles?__hstc=77520074.36a0ddae8e24bce7 (last access, Sep, 2023).

TUMBLIN, J., KORNPROBST, P., PARIS, S. and DURAND, F. *Bilateral Filtering: Theory and Applications*. Now Publishers Inc, 2009.

VICTOR IKECHUKWU, A., MURALI, S., DEEPU, R. and SHIVAMURTHY, R. C. ResNet-50 vs VGG-19 vs training from scratch: A comparative analysis of the segmentation and classification of Pneumonia from chest X-ray images. *Global Transitions Proceedings*, Vol. 2(2), 375–381, Avail. at https://www.sciencedirect.com/science/article/pii/S2666285X21000558?via%3Dihub.

WANG, H., PAN, C., GUO, X., JI, C. and DENG, K. From object detection to text detection and recognition: A brief evolution history of optical character recognition. *Wiley Interdisciplinary Reviews: Computational Statistics*, Vol. 13, 2021. Avail. at https://www.researchgate.net/publication/348781394_From_object_detection_to_text_detection_and_recognition_A_brief_evolution_history_of_optical_character_recognition (last access, Sep, 2023).

Appendix **A**

Appendix A - Contributions

A.1. Cristian Rene Castillo de Leon

Due to the large scale of the project and the ongoing research and implementation work, most of the project has been a collaborative work in which both members of the group have researched and developed together. Even so, I have worked from the beginning with a focus on achieving the project's objectives. Below is a list of my contributions to the project:

- Research on different machine learning technologies and current software testing tools.
  - Created a prototype GameDriver test and implemented it on an Unity proof of concept FIFA 23 menu. This test consisted on a simple point and click test using GameDriver API to reach a destination in the menu. With this test I was able to have an idea on how to proceed with the navigation part of the tool.
  - Recreated the same test on a Unity proof of concept FIFA 23 menu using Sikuli IDE. The visual testing tool helped a lot to define what Cogito needed to do in order to be able to work on multiple types of video game interfaces.
  - Before joining the team, I researched on the capabilities of Amazon Web Services specifically on SageMaker, Rekognition and Textract. Created a few machine learning models and used Jupyter notebooks to follow SageMaker's tutorials.
- Collaborative research into the capabilities of OpenCV and its implementation in Python:
  - Different Canny algorithms to correctly detect the edges in an image. Investigated on the differences between the Otsu Canny, Median Canny and Straight Line Detection algorithms and implemented them as different edge detection strategies for the tool to use.
  - Researched on the Sobel and Laplacian algorithms to detect edges and obtain the sharpness of an image.
  - Applying different types of blur filters to an image to improve the detection of contours. Researched on the Gaussian, Median and Bilateral filters to discard noise and unnecessary elements on the screen.

- Adjusting the contrast and brightness of an image as well as analyzing its contrast and sharpness. Researched on histogram comparison to obtain the contrast of an image.
- Implemented the logic for Cogito's image processing and improved it to be adaptative to multiple types of menus.

- Collaborated in the training of multiple machine learning models to detect and classify icons, texts and buttons using SageMaker and Rekognition. Mainly used Rekognition for the training of icon detection models and specific icon classification. This training involved:
  - The collection of a large number of data, images or texts for training purposes.
  - The creation of tools to automatically generate data or format data to be accepted by the model. Used a graphic annotation tool to define the bounding boxes of each element in an image in order to be used for the training of SageMaker and Rekognition models. Each service needed a specific format '.lst' for SageMaker and '.manifest' for Rekognition.
- Implemented the text detection algorithm to further improve Rekognition's OCR:
  - A solution was needed for the 100 words per analysis limitation, I improved the `TextDetect` class to be able to make multiple calls for an image in the case of a high amount of text.
  - Because Rekognition returns only words and does not form lines or paragraphs, a way to format the words to get the paragraphs and lines that appear in an image was implemented.
- Defined and implemented the Python modules structure for better code management.
- Researched on the possible text classification solutions:
  - Creation of multiple machine learning models for text classification using SageMaker. Developed a tool to train SageMaker text models more efficiently. After much testing, the machine learning model required huge amounts of data to be able to perform correctly, something a video game interface does not have.
  - Implemented the text classification algorithm based on labels of dictionaries and classification strategies.
- Collaborated in the implementation of the detection of aligned groups of elements, focused mainly on the horizontal and vertical groups. This aligned group detection helped on finding specific menu components such as vertical lists, menu bars or input actions.
- Defined the way in which the tool finds the navigable menus:
  - Researched on the histogram comparison and histogram intersection. Implemented with that research the highlighted button detection algorithm. This algorithm was crucial for the tool to work properly since it needs to know exactly where it is currently located on a menu.
  - Defined and implemented the different Checker algorithms to find navigable menus on an image. Combined the highlighted button detection algorithm with the checkers in order to further improve the detection of a navigable menu.

- Defined the way in which the data obtained by both the Lambda function and the local tool are returned for use in navigation. Now the Cogito identification module can be configured to be executed on local or on remote.
- Implemented the configuration module and added debug options to visualize the data in each step of the pipeline.
- Defined and implemented the CogitoApp module, created a menu navigation loop using the data returned by the Cogito module in addition to implementing the input of the tool using the PyDirectInput library.
- Constantly tested the tool with multiple menu screenshots as well as performed full navigation tests with different types of games to prove the effectiveness of the tool.
- Researched on the different ways to detect input actions on the screen. Investigated on a machine learning model to detect input actions which cannot be correctly detected by the OCR.
- Researched on the creation of an executable and also on the creation of a graphical user interface for the Cogito module. This executable would only include the identification and classification of data and it would be a proof of concept for other people to use.

A.2. Iago Quintas Diz

As it has been said on the previous section, this project has been a collaborative work in which both of us have been supervising every step. Along these contributions I have not been the only responsible, but I have been the one more involved in these points. With that been said, my main contributions have been the following:

- Research on existing technologies and other existent solutions that did similar processes than the ones done by our work.
    - Investigation of Tesseract's OCR to detect text on images with plane color backgrounds in real time. A small program was made to determine how well it performed, and that made us understand how OCRs often heavily depend on the font used to understand the text.
        - Research on training Tesseract with different fonts to increase the confidence of custom fonts. Understanding now the problems that OCR face, the decision to use a machine learning solution was made.
    - Training of simple models in Rekognition with a small dataset to familiarize with AWS services before joining the team. This first model helped to understand the formats in which information is provided to the model, the requirements and possible errors that can appear in the process, and the metrics to determine if a model has had good results or not.
    - Creation of small tests on Selenium and Sikuli test frameworks to reproduce certain actions and understand their functionalities better before designing the tool. This helped to adopt an approach more focused on visual testing, although ideas from functional testing where also used.

- Creation of scripts to automate the generation of datasets separated on training and test sets for Rekognition and SageMaker images classification models.
  - Investigation about the use of a graphic annotation tool to obtain labeled files in YOLO .'txt' format. Configuration of default labels to match the ones of our models.
  - Automatic detection of labels used on the previous tool to use them directly, without converting the format to another one.
  - Conversion from '.txt' labeled files to '.lst' for SageMaker and '.manifest' for Rekognition.
  - Use of AWS S3 service to host the images only once. The upload is done automatically, as well as the deployment of the models when they finish training.
- Collaborative work on research about OpenCV and its functionalities:
  - Investigation about the possibility of using simpler solutions such as template matching instead of object detection models.
  - Investigation of more complex image comparison techniques such as feature extraction or histogram comparison, and for what situations they are more suitable for.
  - Research about possible solutions to improve canny edge detection, such as morphological operators, and the implementation of canny edge detection applied to every color channel to avoid losing information.
  - Research about other edge detection solutions like straight lines detectors.
  - Work on contour filtering to avoid saving non closed shapes as tiled buttons.
- Collaborative work on grouping tiled buttons based on their alignment, mainly focused on grids. Later this was useful to group elements into specific objects, such as menu bars.
- Investigation of text classification alternatives such as zero-shot models and other natural language processing techniques. Also investigation about using already existing datasets in order to do fine-tuning to get custom text classification models, instead of building one from scratch. That would require a huge amount of resources that are not reasonable.
  - At the end the text classification models were discarded since keeping a general approach to test different games opposed having a specific trained model for each game.
- Training of object detector and image classification models using AWS Rekognition and SageMaker. At the end, SageMaker models were relegated as possible future work. The reason for that is that Rekognition allowed a faster iteration in training models, while SageMaker costed not only more time, but also better fine-tuning. Rekognition also does this automatically.
  - Training of icon detection models, for which I gathered a large quantity of images and tuned them to improve model confidence to reliable levels. This model only had one label, since it had one specific purpose: finding every icon on the screen, without focusing on their meaning. This included the creation of a script

that enhanced icons so the model could learn faster with less data, performing operations such as changing icon sizes, colors, orientations and backgrounds being some of those.

- Training of icon classification models, which are less reliable than the detection model. This is due to increasing the number of labels and the lower amount of images appearing on the dataset. Gathering images for specific icons was a much more complex task. In addition to this, it is important to highlight that public icon datasets for the specific icons that are needed do not exist.

- Evaluation of the tool, planning the different objectives, designing the methodology to do it and analysing the results after that. More specifically, these were the steps done

  - Adding a simplified telemetry system to the tool to record time events, gathering dataframes for each game tested. Events were sent to a class to perform the time measuring operation, and then each value was stored on their corresponding image and step.
  - After that, the tool run throughout all images. Then. manually classification of images was done to determine true positives, false positives, true negatives and false negatives.
  - With the results obtained, graphics and tables were created in order to visualize better the data and reach conclusions faster about what steps have more issues.
  - Finally, with this information, future work was decided about what steps needed more work to improve and be reliable.