Navegación de interfaz de usuario basada en reconocimiento de imagen y planificación de tareas

UI navigation based on image recognition and task planning

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Aarón Nauzet Moreno Sosa

Director
Guillermo Jiménez Díaz

Colaborador
Claudio de Pasquale

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

Navegación de interfaz de usuario basada en reconocimiento de imagen y planificación de tareas

UI navigation based on image recognition and task planning

Trabajo de Fin de Grado en Ingeniería Informática

Autor

Aarón Nauzet Moreno Sosa

Director

Guillermo Jiménez Díaz

Colaborador

Claudio de Pasquale

Convocatoria: *Junio 2024*

Grado en Desarrollo de Videojuegos

Facultad de Informática

Universidad Complutense de Madrid

27 de mayo de 2024

Dedication

To my father, Jose Manuel; my mother, Lucía; and my little sister, Itziar. Thanks for all the support throughout the years.

Acknowledgments

Thanks to EA and the COE team for giving me such an amazing opportunity. Thanks to Claudio, for his guidance throughout all the steps of the process, and his continuous support even during rough times. Thanks to Blanca, Cris, Ricardo and Iago for all the help provided in both implementation and documentation. And thanks to Guille, for reaching out to me and giving me the possibility of take this once in a lifetime chance, and for his infinite patience even when I may not have deserved it. To everyone that believed in me, thanks, from the bottom of my heart.

Resumen

Navegación de interfaz de usuario basada en reconocimiento de imagen y planificación de tareas

La primera impresión de cualquier software para un usuario siempre está ligada a una *interfaz de usuario* funcional, que no interrumpa su *experiencia de usuario*. Por lo tanto, probar esta parte del software es esencial para el éxito de la aplicación de cualquier desarrollador.

Una de las partes más importantes de la interfaz de usuario es la **navegación**. Una prueba adecuada de esta funcionalidad requiere garantizar que todas las partes de la interfaz sean accesibles y se comuniquen con los menús/submenús previstos. Este trabajo es muy tedioso y requiere mucho tiempo de *testing*, especialmente si el desarrollador lo realiza manualmente.

Con esto en mente, se ha creado una herramienta que permite a los desarrolladores crear casos de prueba para verificar que una interfaz funciona correctamente, utilizando una categorización de los elementos en pantalla mediante reconocimiento de imagen, y un planificador de tareas que usa esta informacion para poder navegar automáticamente por dicha interfaz.

Palabras clave

automatización, navegación, planificación, interfaz, *testing*, *scripting*

Abstract

UI navigation based on image recognition and task planning

A user's first impression of any kind of software is always tied to a functioning *User Interface* (UI), that does not break their *User Experience* (UX). As such, testing this side of the software is essential for the success of any developer's application.

One of the most important parts of the UI is the **navigation**. A proper testing of this functionality requires assuring that all parts of the UI are accessible and communicate to the intended menus/submenus. This work is a very tedious and time-consuming part of testing the software, specially if it is made manually by the developer.

With this in mind, a tool has been made that, employing both an UI and a AI task planner based on these results, lets the developer create custom UI test cases to verify that the designed UI is working correctly.

With this in mind, a tool has been created that allows developers to create test cases to verify that an user interface works as intended, using a categorization of the elements on the screen obtained via image recognition, and a task scheduler that utilizes this information to automatically navigate through the aforementioned interface.

Keywords

automation, navigation, planning, interface, testing, scripting

Contents

List of figures

Chapter 1

Introduction

"Cogito ergo sum"
— René Descartes

1.1. Motivation

As software become bigger and more complex, there has been a rise in concern towards development costs. Far from a few times do projects lose valuable time and resources in trying to maintain and fix the different issues that arise as the product increases in scale, as seen in Figure 1.1.

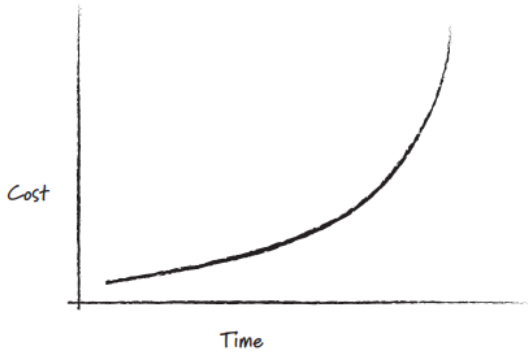

Figure 1.1: Development cost increase over production time (Keith, 2015)

As such, although manual tests are still required during development, it is in any team's best interest to generate automation pipelines for testing. This allows developers to easily test new features without the time consumption that means having to ensure that these features work as intended.

For this reason, different approaches, such as AI-powered testing using generative AI models or neural networks, can be more efficient and cost-effective if implemented with in a testing workflow during the development cycle, where the machine does the repetitive work and the developer assess the results of the tests to improve the product.

One field of software development that greatly benefits of these automation tools

is game development, where the intricacies of the multiple fields involved into its design processes makes it an ideal candidate for automated testing.

One of the most overlooked and sometimes neglected areas of game development is the user interface. Automatic tests for this field can be quite beneficial to the development team, since it ensures menu flow, functionality and feedback are working as intended for every iteration of the interface. Supporting automatic movement actions through these interfaces could help many teams, since it would ease time constraints derived from them, and focus on continue adding features to the final product.

1.2. Objectives

The main objective of this tool is to create an adaptive tool for automatic testing of menus, supported by automatic planning, that can work on multiple different game titles. This project is a continuation of another project (Castillo and Quintas, 2023), which was in charge of a tool, named Cogito, that is able to recognize and classify different elements from a game screenshot, like icons, text and menu structures.

Initially, an investigation of the tools currently available for user interface testing will be conducted, focusing on what level of knowledge about the interface structure the program requires, what set of navigation actions they support, and what other aspects may be useful in order to navigate menus. In addition to this, a special emphasis will be placed on the investigation of AI techniques that may allow the tool to successfully automate navigation without any information aside from what visible on the screen, that is to say that no data structures will be available beforehand.

After understanding the advantages and disadvantages of other tools, the navigation tool will be developed to allow automated navigation, and an iterative process will be followed to build the tool.

Once the tool is built, its performance will be evaluated by checking the time performance. It is crucial that the decision-making process for navigation does not add more cost to the already developed image recognition module, otherwise its use would be undesirable.

The project will serve as a tool to improve automatic interface testing for Electronic Arts, allowing testers to just lay down the guidelines of any user interface test for any game without the need of adjustments between game titles. This delegates the monotonous and repetitive tasks to the automated software, which will enhance work productivity and provide an opportunity to focus on more complex tasks.

1.3. Work plan

As part of the Electronic Arts developer pipeline, this project will be developed following the SCRUM guidelines (Pichler, 2010), an agile product management methodology consisting of short iterations, called "sprints", which ensure that there is a working version of the product readily available, that demonstrates the progress

made. Given these guidelines, the work plan will be divided into two-week sprints, where a series of tasks will be assigned to the developer. These tasks are valued given their complexity, and have a priority to them, in order to indicate which tasks should be completed before others. At the end of each sprint, resolved tasks will be closed, the results will be shown to the team, and a new sprint iteration will begin. The number of tasks managed is expected to start at a slow pace and escalate in accordance to the progression of the tool's development. The team uses an in-house version of Jira in order to handle sprint management.

Before the start of the developing cycle, a research will be conducted to ascertain which techniques could be deployed to achieve the results that the tool has set itself to obtain. In doing so, there will be an extensive research of the image recognition pipeline, to understand all its intricacies. Then, different AI task planner and formal grammar libraries will be considered to use for the different behaviours of the tool.

Although Python will be used in the earlier stages of development, to test both the pipeline and to try different approaches of the planner behaviours, the team is in the process of migrating to C#, so this language, along with Microsoft Visual Studio as the environment, will be used in the end product. In order for the project to be manageable, Git will be used as the version control system, and the repositories will be hosted in GitLab.

Throughout the development phase, the code will be subject to inspections to improve both the tool's efficiency and performance, in order to comply to industry's standards.

1.4. Document outline

This document is divided in the following chapters:

1. Introduction, where the project's objectives and motivations are outlined.
2. State of the art, where the concept of formal grammars, graph theory, and different AI techniques are explained, and different tools alternatives for each of the subjects discussed are discussed.
3. Tool design, where the first approach about the tool behaviour, features and structure are described, assuming an ideal development cycle.
4. Tool development, where both the initial work and prototypes, as well as the development iterations are depicted, as well as how the tool structure is delineated into each subsequent module.
5. Evaluation, where the results of the tests applied to the tool are reported.
6. Conclusions and future work, where the results of the project are discussed, along with any possible future work that could come out from it.

Chapter 2

State of the Art

Before starting the development of the navigation tool for user interface testing, a research has been conducted prior to any design or development decision. Thus, this chapter discusses the current state of UI testing, with a software example that fulfills this task.

After considering the limitations that the currently available tools have, a research was conducted in 3 fronts. First, in order to be able to automate the user interface navigation, different AI techniques and how they operate were explored, such as AI planners and finite-state machines. Second, to create a custom scripting language that allows users to generate their own navigation tests, formal grammars were researched. Finally, in order to better represent the UI structure internally, graph theory and graph representation were investigated.

How all these concepts work will be studied in depth across this whole chapter.

2.1. UI testing

UI testing is a particular field of software testing where developers focus their efforts in searching and fixing errors present in the user interface of software applications, in order to not disturb the user experience related to it. Although it is a task that can be handled manually, the nature of iterative development that software is usually constrained to means that, in most cases, any development team wants to perform these tasks automatically, as it saves time and resources.

Based on the aspects being tested in the UI, this field of testing can be divided into two categories: **functionality**, where it is ensured that any element present in the UI performs all their expected behaviours correctly (i.e., raising/lowering the volume when moving the corresponding slider); and **visuals**, where the elements of the screen are tested to ascertain that they are correctly organised, sized and coloured according to their behaviours, while also giving the right amount of feedback given any state they are in (i.e., have a button change to a darker color and a bigger size when hovered).

In order to be able to properly test all these issues, UI testing systems require different actions to be executed. These actions usually come with a series of specific conditions that need to be met before continuing the action sequence defined by the tester.

Depending of the machine these tests are running on, specific support is needed to fulfill these conditions, and this is where automatic testing software comes in. One of the most commonly used tools currently available that development teams use to automatize all these tests is Selenium.

2.1.1. Selenium

Selenium (Huggins, 2004) is a framework for automatic website functional testing. It allows to create scripts for web navigation, supporting common web actions like clicking elements or filling forms. Selenium is divided in two tools, each of them with a different function:

- **WebDriver**: an API that defines a neutral interface to control the behaviour of web browsers. It uses browser-specific automation APIs provided by each browser vendor to build drivers that act as a bridge between the script and the browser.
- **IDE (Integrated Development Environment)**: allows to develop Selenium test cases. It exists as a Chrome or Firefox extension, due to that in this way it can directly operate on the browser. As Figure 2.1 shows, the IDE allows the creation of scripts, a list of commands that will be performed in order.

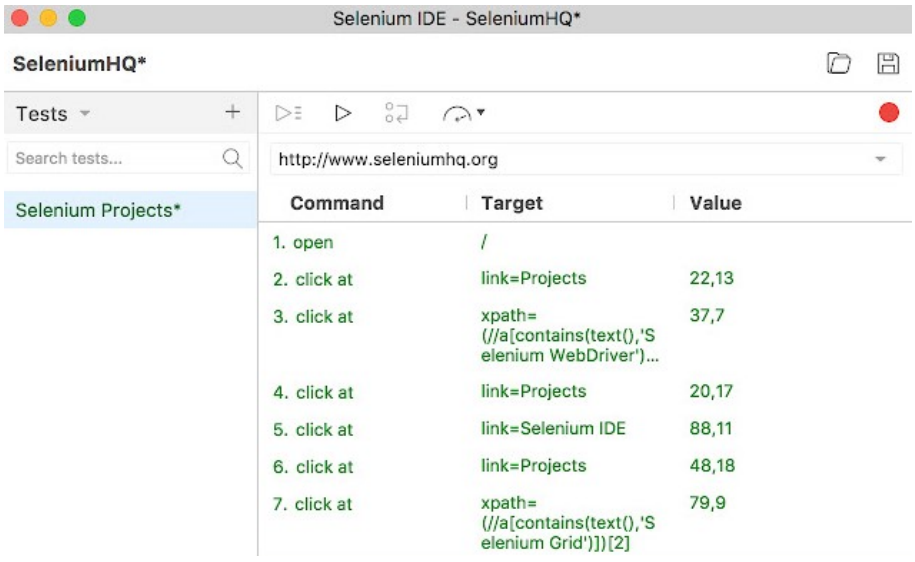

Figure 2.1: Selenium IDE

To be able to test as many functionalities as possible, the Selenium scripting language supports three different groups of commands:

- **Input actions**: since Selenium can access the HTML source code, it can be able to perform actions on any element by looking for their identifier or properties, like clicking on a certain button.
- **Waits**: as the tests will depend on external factors like the machine's internet connection, they will need to be delayed artificially until certain conditions apply, like a certain element appearing. For that purpose, both active waits, where a delay of a specified amount of seconds happens, and waits to an element to appear on screen are supported.
- **Asserts**: to be able to check if a test has been success, there are multiple instructions to verify the correct state of the webpage, like if an element is present or if an specific word is located in the page.

2.1.2. Airtest

Airtest (NetEase, 2014) is a cross-platform automated testing framework for games and applications. The Airtest framework is designed to leverage image and text recognition technologies for testing across multiple platforms, like Android, iOS, Windows, and Unity games.

Airtest provides an IDE, that makes it possible to write and execute test scripts, using Python as its scripting language. The Airtest framework also includes Poco, a user interface automation framework designed to automate applications at the UI level, with commands like **waits**, **asserts**, and actions typically done for mobile phones, such as **swipes** and **touches**.

As seen in Figure 2.2, the framework supports image recognition using OpenCV, which allows users to define commands that use images as a value, like the ones supported by the Poco API.

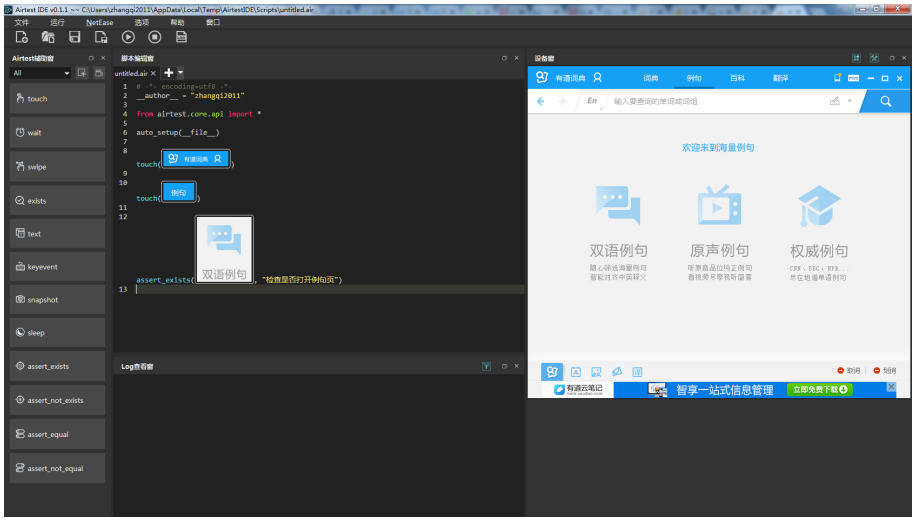

Figure 2.2: Airtest IDE

2.2. AI Planning

One of the many branches of artificial intelligence is AI planning (Vallati and Kitchin, 2020). This research field is usually in charge of exploring how to deploy autonomous techniques to solve scheduling problems. These problems try to synthesize a bulletproof plan in order to arrive at the desired goals using the available actions at one's disposal. This technique has many applications in the industry, like the creation of robots, autonomous systems and cognitive assistants, or for cybersecurity issues.

The complexity of these problems rely on what assumptions are made beforehand. Although the planning problem can get more complex depending on different variables, like if the actions are deterministic, or they have any duration to be done, the simplest type of planning is defined by:

- a unique known initial state.
- durationless deterministic actions, that can only be taken one at time.
- a single working agent

In this kind of planning problem, the plan can be established as a series of actions, that can be easily predicted given the previous conditions. These specific type of problems are usually classified into classical planning. Proper algorithm used in order to solve classical planning are forward chaining and backwards chaining state space search, methods commonly used in rule-based systems for decision-making.

Forward and backward chaining are logical inference methods that obtain a solution by using defined rule systems to extract data and arrive at the desired goal. In a state space search, forward chaining would take the initial state and, given the rules defined by the possible actions of the planning problem, asses new data until a goal is reached. Meanwhile, backward chaining would do the opposite, starting the goal and working backwards to find supporting actions, until a known initial is reached.

It is common for planners to define a domain model, which is outlined by the potential actions to be deployed, alongside the specific problem to be solved, inferred by the initial state and goal. The most common used languages to representing these domains are STRIPS and PDDL, since they are based on state variables. In these languages, each state is an assignment of values to the state variables, and actions are in charge of how these values fluctuate when any of them are executed. Figure 2.3 shows how a domain and problem can be defined for a construction planning problem, with the action of building walls, and the goal of completing the walls for the building.

Although PDDL is a very useful tool to use for common software planning, it does not translate very well to game development. As such, different planning architectures have been developed, one of them being GOAP, which will be discussed in the next section.

```
(define
 (domain construction)
 (:extends building)
 (:requirements :strips :typing)
 (:types
 site material - object
 bricks cables windows - material
 )

 (:predicates
 (walls-built ?s - site)
 (windows-fitted ?s - site)
 (foundations-set ?s - site)
 (cables-installed ?s - site)
 (site-built ?s - site)
 (on-site ?m - material ?s - site)
 (material-used ?m - material)
 )

 (:action BUILD-WALL
 :parameters (?s - site ?b - bricks)
 :precondition (and
 (on-site ?b ?s)
 (foundations-set ?s)
 (not (walls-built ?s))
 (not (material-used ?b))
 )
 :effect (and
 (walls-built ?s)
 (material-used ?b)
 )
 )
)
```

```
(define
 (problem buildingahouse)
 (:domain construction)

 (:objects
 s1 - site
 b - bricks
 w - windows
 c - cables
 )

 (:init
 (on-site b s1)
 (on-site c s1)
 (on-site w s1)
 )

 (:goal (and
 (walls-built ?s1)
 )
 )
)
```

Figure 2.3: PDDL construcion domain (black) and problem (grey) definition

2.2.1. Goal-Oriented Action Planning

Goal-Oriented Action Planning (Rabin, 2015) is an automated planning architecture particularly useful in game development, where it is used to create autonomous agents, like non-playable characters (NPCs), that dynamically plan a sequence of actions to satisfy a goal.

Similarly to planning problems, the GOAP architecture consists of the following elements:

- A series of **states**, defined by conditions that can be achieved by the agents or by the environment.
- A **goal**, usually in the form of a condition that has to be met either in the agent or in the environment state (i.e., agent health reduce to half of its total amount).
- A series of **actions**, that each agent are able to perform, These actions have to be atomic, encapsulated and ignorant of the other actions available. Each action has both preconditions and effects, that allow themselves to be chained into a plan.
- A **plan**, a sequence of actions that satisfy a goal from an agent's starting state conditions.

This structure makes the codification to be more modular an easier to maintain, by isolating states from each other, in comparison to other techniques like FSMs, that, as explained in section 2.3, require states to be linked to each other. In Figure 2.4 this simplification is showcased visually.

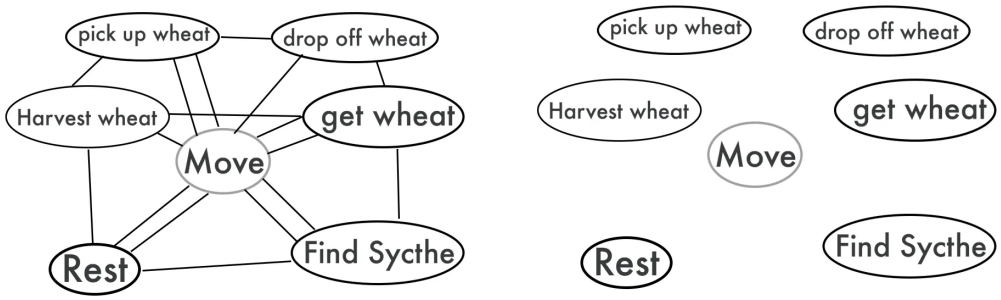

Figure 2.4: FSM vs. GOAP representation

The GOAP process to create a plan is commonly referred to as "formulating a plan". In this process, the planner looks at every actions preconditions and effects, and queues them in order, to satisfy the goal given the world state. The planner locates the solution by building a tree, created by running through every possible action using pathfinding algorithms, to then obtain the most optimal solution to arrive at the target goal.

A lot of games have employed the GOAP architecture in their games, to define complex game AIs. The one game pioneering the use of this system was famously F.E.A.R (Orkin, 2006), which uses this technique to define the behaviour of their combat NPCs.

2.3. Finite-State Machine

A finite-state machine (FSM) (Mihov and Schulz, 2019) or finite-state automaton (FSA), is a mathematical model of computation. An FSM is represented by an abstract machine that is composed of a list of states, transitions, and inputs. This machine can only be in exactly one of these states at any given time.

To explain each element in FSMs, a door locking system will be used as an example:

- **States** are a representation of a specific condition or situation that the system can be in. For example, a door would have the states "locked" and "unlocked".
- **Transitions** define how the system changes from one state to another in response to external or internal events. For instance, when a key is used in a door, it transitions from the "locked" state to the "unlocked" state.
- **Inputs** are the event that FSMs use to trigger state transitions. In the door locking system, the input would be the key.

FSMs have their own classification based on their transition rules: **deterministic** FSMs (DFA), where there is a fixed set of rules for transitioning between states, allowing for just one state to be the next one given any current state and input; and **non-deterministic** FSMs (NFA), where there are multiple possible next states for the same state and input. However, any DFA can be constructed to be equivalent to any NFA.

Thus, a deterministic finite-state machine is formally defined as a quintuple $(\Sigma, S, s_0, \delta, F))$, where:

- Σ is the input alphabet, consisting of a finite non-empty set of symbols.
- S is a finite non-empty set of states.
- s_0 is an initial state, which has to be an element of S.
- δ is the state-transition function $\delta : S \times \Sigma \to S$. It is commonplace to allow δ to be a partial function, meaning that not every pairwise combination between an element from S and an element from Σ has to implement a transition.
- F is a, possibly empty, subset of S representing a set of final states.

2.4. Grammars

A formal grammar (Johnson and Zelenski, 2012; Meduna, 2014) refers to a set of rules by which valid sentences in a language are constructed, using different forms and combinations of words to make these sentences structurally correct. In computer science, these grammars are usually applied in the creation of *custom programming languages*, based on the applied mathematics field of formal language.

Any formal grammar is built upon these basic principles:

- **Terminal** symbols: each static word in the language that can not be replaced by any other symbol, acting as a dead-end.
- **Non-terminal** symbols: any other grammar symbol that allows for its replacement or expansion into any sequence of symbols.
- **Productions**: each rule defined in the grammar that explains the process in which each symbol can be replaced or expanded upon. The application of these rules on a grammar symbol will devolve into constructing syntactically correct and complete *sentences*, which are strings consisting only of terminal symbols.
- **Derivations**: each rule defined in the grammar that explains the process in which each symbol can be replaced or expanded upon. The application of these rules on a grammar symbol will devolve into constructing syntactically correct and complete *sentences*, which are strings consisting only of terminal symbols. These derivations will be referred to in the future as **parses**.

And so, it can be formally stated that a grammar is a tuple (S, P, N, T), where P, N and T refer to a finite set of production rules, terminals, and non-terminals respectively; and a start symbol S (also known as the sentence symbol).

In order to represent the derived parses, a grammar will usually employ a parse tree, where each symbol is linked to others in a hierarchical manner.

For example, for the sentence "This is a university", where the productions define it as a combination of a subject, a verb and an object, a parse tree would be akin as the one showcased in 2.5.

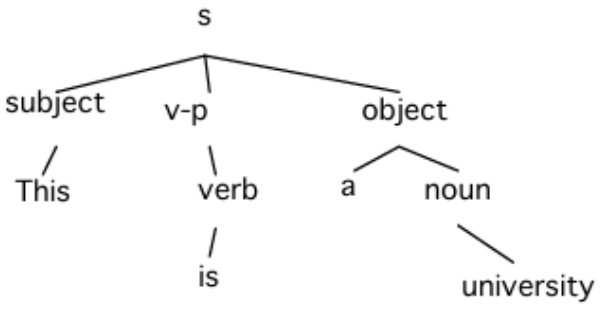

Figure 2.5: "This is a university" parse tree

The construction of a grammar raises the question of how efficiently it can be parsed without losing expressiveness. This question has created a classification of them in four different groups, depending on the restrictiveness of the productions in place:

- **Free/Unrestricted grammars**: there are no restrictions on what appears on the left or right side of any given production, aside from the fact that left hand side must be non-empty, making these grammars the most general ones and, in turn, the most inefficient in terms of parsing.
  Production: $u \rightarrow v$, where both u and v are arbitrary strings of symbols, with u non-null.
- **Context-sensitive grammar**: in these grammars, a symbol can be replaced only when the specified context is in place.
  Production: $uXw^{\smile} > uvw$, where u, v and w are arbitrary strings of symbols, with v non-null, and X a single non-terminal.
- **Context-free grammar**: in this case, you can replace any symbol with what the grammar rule assigns them to, regardless of context.
  Production: $X \rightarrow v$, where v is an arbitrary string of symbols, and X is a non-terminal.
- **Regular grammars**: these grammars require that the left side has to be a single non-terminal, and the right side can be either empty, a single terminal by itself or a single terminal paired with a single non-terminal. This makes them the most limited in terms of expressive power.
  Production: $X \rightarrow a \vee aY \vee \epsilon$ where X and Y are non-terminals and a is a terminal.

Given these concepts and constraints, the industry uses formal grammars for many complex issues, such as natural language processing or creating custom programming languages. To be able to accomplish this, developers need some kind of

tool to define and use these custom formal grammars for their programs. As such, the following sections will layout some tools that can be used for this purpose.

2.4.1. Grammatical Framework

Grammatical Framework, or GF (Ranta, 2011), is a functional programming language heavily based on Haskell and ML that supports the implementation of multilingual grammar applications.

GF is able to efficiently parse natural languages, allowing developers to implement custom definitions of grammatical rules, lexical and phrasal rules; syntactic combinations, inflection tables and paradigms to handle word forms for each grammar.

The GF programming language brings features like static type checking, higher-order functions or a module system with multiple inheritance. It also comes with a command interpreter and batch compiler, allowing the compiled parser and translator to be embedded into applications written in other languages, like Java, C++, Haskell or JavaScript.

2.4.2. ANTLR

ANTLR (ANother Tool for Language Recognition) (Parr, 2012), is a parser generator that assists in transforming raw input data into a structured representation.

Although ANTLR Java to make its build, it does not mean it is only available to use in that specific language. The different characteristics that ANTLR provides can be targeted to be employed in a plethora of languages, like C++, C#, Java, JavaScript, PHP or Python.

In order to achieve this, the ANTLR build provides the following functionalities:

- A **lexer/tokenizer** that allows to break the raw input text to be parsed into smaller units.
- A **parser** that grants the developer the ability to set different parse rules, using the tokens defined by the lexer. Then, these rules are applied to construct the corresponding parse tree, as showcased in Figure 2.6. This resulting parse tree is referred in ANTLR's API as an Abstract Syntax Tree (AST).
- A **listener (or visitor) interface** which facilitates the addition of behaviours depending on which place inside the AST we are currently listening/visiting.
- An **error handling** mechanism to detect syntax errors during runtime.

For these reasons, the tool is widely used in the industry in order to build different languages, tools, and frameworks. One big example is its use in X (formerly known as Twitter), where they use it to parse all queries coming from the page's search bar.

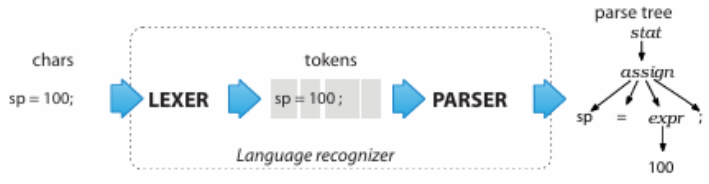

Figure 2.6: ANTLR parse pipeline

2.5. Graphs

In discrete mathematics, graph theory (Dharwadker and Pirzada, 2011) refers to the study of graphs, mathematical structures utilized to illustrate pairwise relations between objects. A graph is a group of elements, known as **nodes** or **vertices**; linked between each other by a series of connections known as **edges**.

Thus, a graph is formally referred to as an ordered pair $G = (V, E)$, where V is a set of vertices; and E a set of edges, each of them associated with two distinct vertices (that verifies $E \subseteq \{\{x, y\} \mid x, y \in V \wedge \ x \neq y\}$).

Depending on the type of connection these edges represent, graphs can be categorized as undirected graphs, where edges represent a symmetrical link, or directed graphs (also known as digraphs), where they represent an asymmetrical, one-way link. Also, these edges may contain weighted values, that represent the cost of their corresponding edge. These categorization can be visually discerned in Figure 2.7.

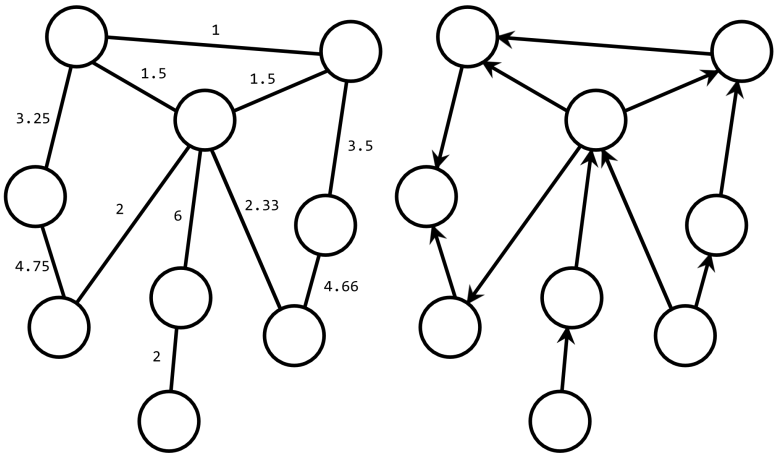

Figure 2.7: Example of undirected weighted graph (left) & unweighted digraph (right)

Graph theory has applications in various fields such as chemistry, physics or topology, as a way to represent different structures or interactions. In computer science, graphs are employed in different algorithms and data structures, which has led to a need to define how a graph should be represented, which many developers have tackled over the years. One of the most common representation challenges are related to hierarchical graphs, a problem that the following formats try to solve.

2.5.1. GEXF

GEXF (Graph Exchange XML Format) (Bastian et al., 2009b) is a XML format developed with the purpose of describing complex networks structures, their associated data and dynamics. As part of the Gephi project, GEXF has been specifically designed to work in conjunction with Gephi (Bastian et al., 2009a), an open-source visualisation tool for networks and graphs, allowing an easier visual representation for raw text files as shown in Figure 2.8.

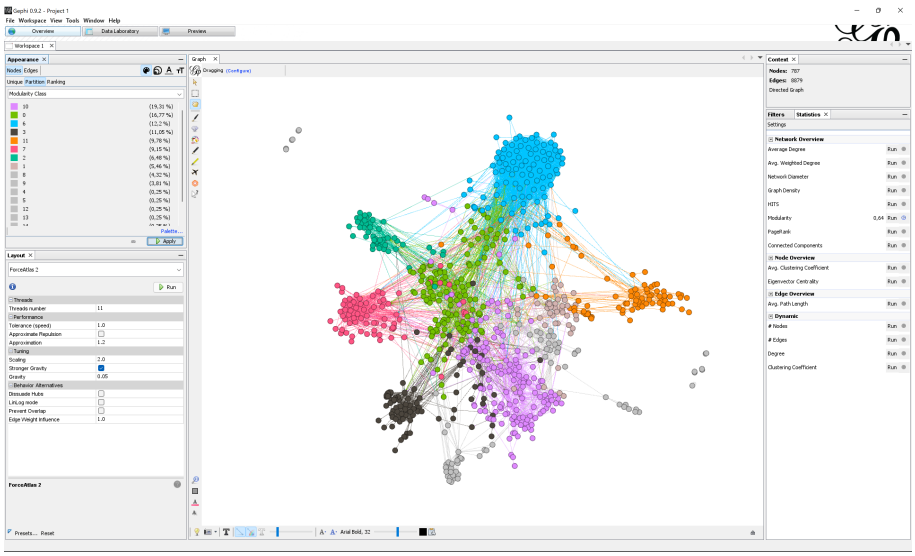

Figure 2.8: Gephi network visualisation

What differentiates GEXF from other graph formats is its specialisation on representing networks. For that purpose, GEXF allows both network dynamics that saves modifications through time, and hierarchical structure to better represent clustering. In order to represent hierarchical structures, GEXF has three alternatives:

- **Node nesting**, where nodes can be present inside other nodes that allows to break the raw input text to be parsed into smaller units.
- **Parent identifier assignment** as a parameter of any node.
- **Phylogeny**, where nodes will save the information on any ancestor they have. This allows for multiple nesting without needing to duplicate information.

2.5.2. GraphML

GraphML (GraphMLTeam, 2002) is an XML format that allows comprehensive file formatting for graphs. Its composed of both a language core that reports structural properties of graphs, and a flexible extension system to add application-agnostic information.

To satisfy this, GraphML contains many features, such as support of directed, undirected, and mixed graphs; hypergraphs, hierarchical graphs, graphical representations and application-specific attribute data; along many more.

2.6. Conclusions

This chapter has showcased the current state of user interface test automation. In doing so, to be able to develop a tool for automatic navigation across multiple games and platforms, many fields were researched afterwards.

Different AI techniques, like AI schedulers and finite-state machines, were explored to automate navigation through different user interfaces. Also, a research into how formal grammars function and how are they used in the creation of custom programming languages has been conducted. Finally, graph theory and representation was done, in order to be able to properly represent the internal structure of a user interface. This investigation has given a lot of insight into what tools are commonly deployed for each of these fields, and how to use them in this project.

After this study, the next chapter will tackle the design of the tool in depth.

Chapter 3

Design of a tool for automatic user interface navigation testing

During the research phase, different tool solutions to automatic user interface navigation were investigated, in order for the team to fully understand the current state of automated user interface testing. A key element in almost all of them is the presence of an internal representation of the user interface, readily available to be used to select navigation movement. Also, in these tools specific platform-agnostic actions are required for the tests to be executed, which have to be specified by the user and, subsequently, require changes if any part of the navigation flow is modified.

With that in mind, the design of a tool that can circumvent these constraints, allowing for the creation of tests that can be applied across multiple games in multiple platforms with minimal to no adjustments, will be explain throughout this chapter.

3.1. Problem statement

To showcase the problem at hand, the following test case will be used:

The user wants to go to the display settings menu, and afterwards change the display mode to fullscreen mode.

In this case, the Selenium IDE, as seen in subsection 2.1.1, would require the developer to define a sequence of actions manually, and, using the internal HTML DOM, would execute these actions until it arrives to the end. Meanwhile, in the Airtest IDE, as seen in subsection 2.1.2, the same sequence of actions would be defined, but this time using previously made screenshots to be able to decide where to go on the screen.

However, if a game developer used a tool with a similar behaviour, many problems arise:

- If the developer does not know how the navigation flow works internally, they would need to check the software navigation flow manually, which in many cases (unless they are the one in charge of the internal representation of the software) would entail a loss of resources to know what it is to test, specially

if the elements that would easily point towards the path to follow are hidden or not properly designed.

- In order to test the same functionality in multiple games, and given that they may have drastically different user interface structures, it would require a specific test to be done for each of the games, even though the behaviour to be tested is the same.
- Whenever an unexpected window shows up, like a pop-up which only appears the first time the software is run or an connection error message, the developer would need to create a different test that accounts for that window to appear, meaning unnecessary test duplication.

Given these issues, the objective of this tool is to create a test environment where the user is able to define navigation tests in a generalist way. What that entails is, by defining a singular type of test where the user is testing a specific functionality of the interface, it can be applied to different games, without the need to make certain modifications to fit the test for each of the games' environment.

Furthermore, unlike what other tools have available, the tool would work with the preconception that neither a processing nor a navigation through the interface has been done beforehand, meaning that the tool works without an internal representation provided by the developer, only creating one itself during execution if needed, and saving developers time in the process.

In order to explain them, the previous example showed earlier will be applied on both the F1 22 (ElectronicArts, 2022) and Battlefield 2042 (ElectronicArts, 2021) games:

- In this menu, where no element suggests connection to the display settings menu, the tool would send an error, informing the user that there is not enough data to continue the navigation.

  In Figure 3.1 no element appears to send the user to neither the display settings menu nor anywhere where the tool could arrive to the proper path.

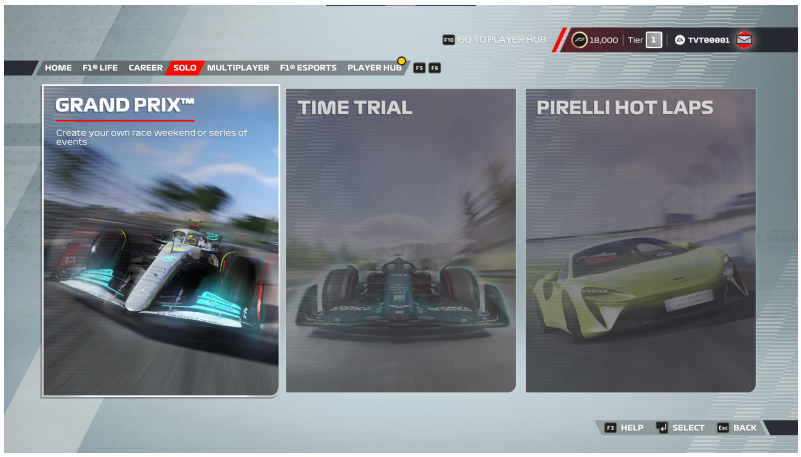

Figure 3.1: Menu case with no element suggestion navigation towards "Display Settings"

- In the case that the current menu is in a situation where it appears to not have an element appears that goes to the objective (in this case, the display settings menu), but an element suggests that, by using it, the path could end up arriving at the desired menu, the tool would execute the actions to move through the UI elements, until it reaches this element and selects it. This step would repeat until either it finds an element that appears to go to the desired menu, or it gets to a menu where no element suggests the required information.

  In Figure 3.3, in both F1 and Battlefield cases there is any element that sends directly to the display settings menu. However, the element "Game Options" in F1, and the element "Options" in Battlefield, suggest that, by going through it, the tool would eventually arrive to the correct menu, given how the display setting menu is usually contained inside the general settings.

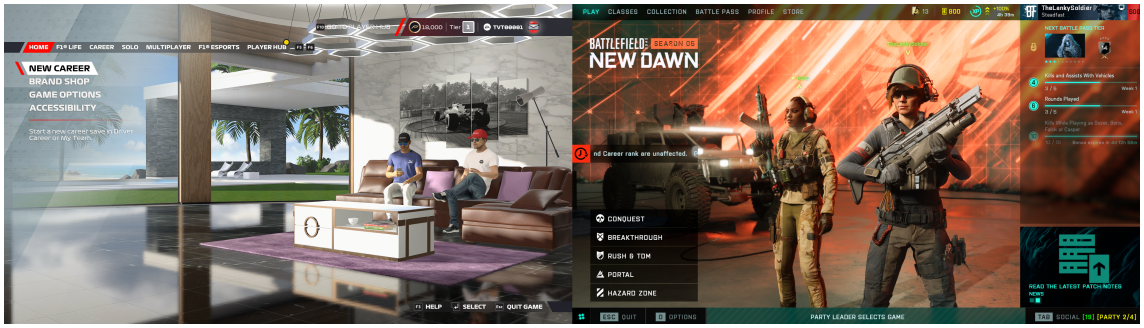

Figure 3.2: Menu with element suggesting access towards "General Settings": "Game Options" (left) and "Options" (right)

- Consequently, if the tool arrives at a menu where an element appears to have a direct connection to the objective (in this case, the display settings menu) the tool would act similarly as in the last case, ending the first step of the test case.

  In Figure 3.3, there is not any element that sends directly to the display settings menu, but the element "Game Options" suggests that, by going through it, the tool would eventually arrive to the correct menu.

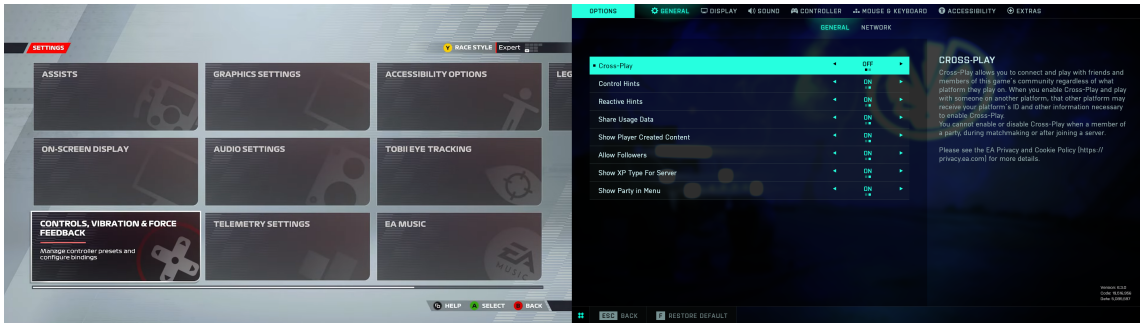

Figure 3.3: Menu with element suggesting access towards "Display Settings": "Graphic Settings" (left) and "Display" (right)

- If the tool has already arrived to the desired destination, via doing multiple steps of the previous situations, it will now search for the specific element to

be modified, and execute a sequence of actions similar to the ones done before, but in this case in order to modify the value requested.

In Figure 3.1, the element "Display Mode" is visible, so the tool would simply move towards that specific element, then execute the actions required to modify the value to "Fullscreen".

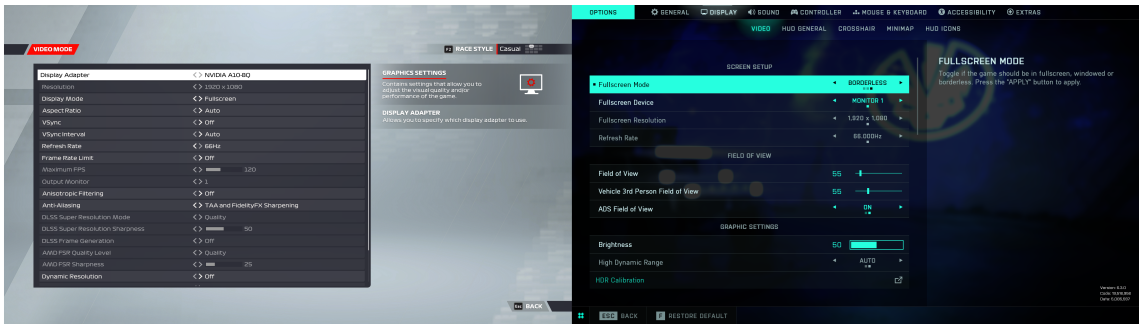

Figure 3.4: Already on "Display Settings", "Display Mode" present (named "Fullscreen Mode" in Battlefield)

In order for the tool to be able to properly solve this issue, the complete tool pipeline, along with its behaviours, has to be defined before any development decision can be made. This will be discussed in the following sections.

3.2. Navigation tool prototype pipeline

This project aims to be an adaptive tool for automatic testing of menus on multiple different game titles. As such, before doing any design decision for the navigation tool, the pipeline was defined, behaving as follows, as seen in Figure 3.5:

1. The user creates a suite of navigation tests on any supported type of file, which will then work as the entry point of the software. These test files will be processed by the **scripting module**, saving them internally in order to be executed.
2. Afterwards, their commands will be run sequentially. In order for the test navigation commands to work, three different modules will be working together to achieve its goals:
   - The **perception module** will be in charge of taking the necessary screenshots of the game, and then recognizing and classifying the current elements present on the screen.
   - Then, the **decision-making module** will use the data provided by the perception module and define a sequence of actions that are needed to be executed for the navigation command to work.
   - Finally, upon deciding what actions to execute, a series of **actors** will be in place, that will help the decision-making module to execute the required actions necessary for the test.

This includes both an input module, that allows the abstraction of platform-agnostic controller actions, so the decision-making module can act independently from the platform tested; as well as the graph module, which will be in charge of persisting the internal navigation structure of the user interface whilst the navigation actions take place, in order to use this data in future runs of the software, to save computing time.

3. All the tests created would then be executed, outputting the final results to the user, either via the assertions that verify that a test has failed or passed, or sending the necessary error messages when there is not enough information for the decision-making to continue its job.

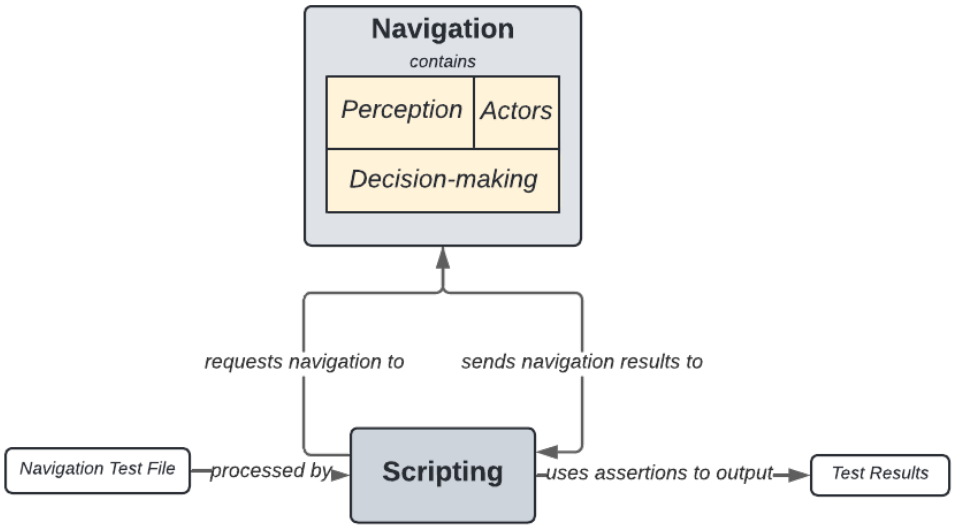

Figure 3.5: Navigation tool pipeline prototype

3.3. Navigation tool design

After laying out every single aspect that the tool's pipeline is trying to tackle, the general definition of each of the modules present in the pipeline was next, in order to successfully develop them in the future

Before doing that, it is important to explain some concepts first. Although the tool does not have an internal representation of the user interface structure, the information provided by the image recognition module has to be categorized in some way. As such, the tool will work with the concept of a **state**.

The tool will consider a state to be any layout/menu that has a significant difference to the other layouts/menus that make up the software's UI. For example, as seen in Figure 3.6, the first two screenshots in the upper row would be considered the same state, since, although there are different UI elements present in the second one, they both share menu structure and the majority of the elements. Meanwhile,

the next two screenshots in the lower row would be considered different states from each other, given their stark difference in both UI elements and menu structure.

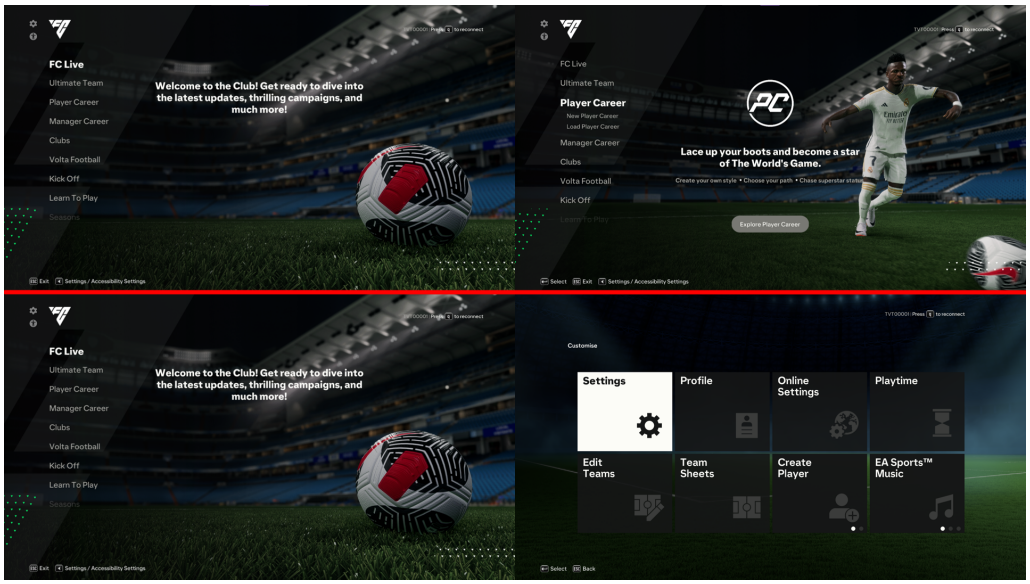

Figure 3.6: Difference between two screenshots considered the same state (upper row) and two screenshots considered different states (lower row) in EA SPORTS™ FC 24

Another issue to consider are **dynamic states**, states that are not present through all instances of the software tested (i.e., EULA pop-ups). The main objective will be to skip them in order to resume navigation and avoid unexpected shut-downs.

The following sections will describe how each of the new modules explained in section 3.2 will work.

3.3.1. Scripting module

To support the creation of personalised navigation test cases by the user, a custom scripting language for the tool will be developed, using a formal grammar as explained in section 2.4, that will allow developers to utilise the different supported commands to ensure that the UI they have built behaves as intended.

As such, the custom scripting language will be developed to support the following instructions:

- **test *testName*:**: entry point for the test *testName*.
- **goto *state***: reach the state specified by the user.
- **modify *uiElement newValue***: move to an element in the current state, and modify its value to *newValue*.
- **assert *assertValue***: ensure that the *assertValue* is present in the current state.
- **end**: exit point for the test.

Given this information, the following is an example of a plausible test for the navigation tool's scripting language:

```
test case1:
goto graphics
modify "display mode" "fullscreen"
assert "fullscreen"
end
```

3.3.2. Perception module

As it was mentioned in section 1.2, this tool is an extension of the image recognition done in a previous project (Castillo and Quintas, 2023), and so, the perception module will encapsulate this behaviour in its entirety.

Although the discussion on the implementation of this module goes out of the scope of this project, the only critical thing to note is that the results of processing each game's screenshots via image recognition techniques will result in a `.json` output, that will provide every single UI element detected with their metadata, such as their bounding box or the generic labels that approximate the expected behaviour of them upon use. All this information will then be used by the decision-making module, which will take the data of these UI elements and take navigation decisions accordingly.

3.3.3. Decision-making module

After processing the instructions given by the user through the scripting language, the program has to decide what sequence of actions to follow regarding navigation. After a research towards different techniques that would allow this module to automate this process, it was decided to use an AI scheduler to handle the decision-making; more specifically, a Goal-Oriented Action Planning architecture will be deployed.

In subsection 2.2.1 it was mentioned what components compose each planning case in a GOAP architecture: **agent**, **initial agent state**, **goal** and **possible actions**. In terms of available abstract actions, there are two distinct types that the planner can consider using:

- **Navigation movement actions**. These include the *left*, *right*, *up*, and *down* actions that are usually employed to navigate through menus in games.
- **State change actions**. These include both *select* and *back* actions, which will go forth or backwards through the navigation flow, respectively; as well as the *input action*, a special type of action that entail the press of a button to move to another menu specified by their assigned tooltip.

Now, at any given planning moment, the planner needs to fulfill the following three goals to execute a navigation command:

- **Preprocessing**. During the planning of this goal, a series of actions will process the output receive from the perception module, filtering the useful information for the planner to decide the sequence of navigation actions. This information ranges from filtering the required labels used in the perception module, as well as locating the menu structure most likely to be the main menu, and the UI element that the planner would like to move towards to arrive at the desired state specified by the user.
- **Movement**. Once all the preprocessing is done, this goal will plan towards obtaining the desired sequence of navigation action to execute, in order to move through the user interface. If the desired state is not achieved by this point, it will loop to the preprocessing again.
- **Finish command**. Once the desired state is reached, every navigation command has the possibility to execute a final action to end their behaviour. For example, during the modify command, once the element is reached, it will try to change its inherit value to the one specified in the command.

Alternatively, if the planner has a persisted representation of the user interface of the game being tested, and can ascertain that a path exists between the current state and the one specified by the user, it will skip this process, opting to obtain this path and executing it directly, saving time in the process.

3.3.4. Actors module

This module encapsulates extra functionalities needed for the complete pipeline to function as designed, and are not the responsibility of the other modules previously described.

3.3.4.1. Input module

For the decision-making to apply the corresponding actions described previously, some back-end work has to be made in order to support inputs. Given what the tool aims to do, the **input** module needs to isolate any platform-agnostic action, so the tool can make general movement decisions that are not tied to any platform supported.

As such, for the design of this module, an input manager similar to the one used in the Unity engine will be developed, letting the user extend each of the possible movement actions with multiple options using different “input layers”, referred to as "axis" in Figure 3.7. With this, the platform-agnostic technologies can be wrapped, allowing the user to work on multiple platforms and also being able to be readily expanded to accommodate additional platforms in the future.

3.3.4.2. Graph module

During the movement step of the decision-making module, although no connection graph is given beforehand, a navigation graph can be built during runtime, and persisted in a file format that is able to be easily stored and processed, as seen in

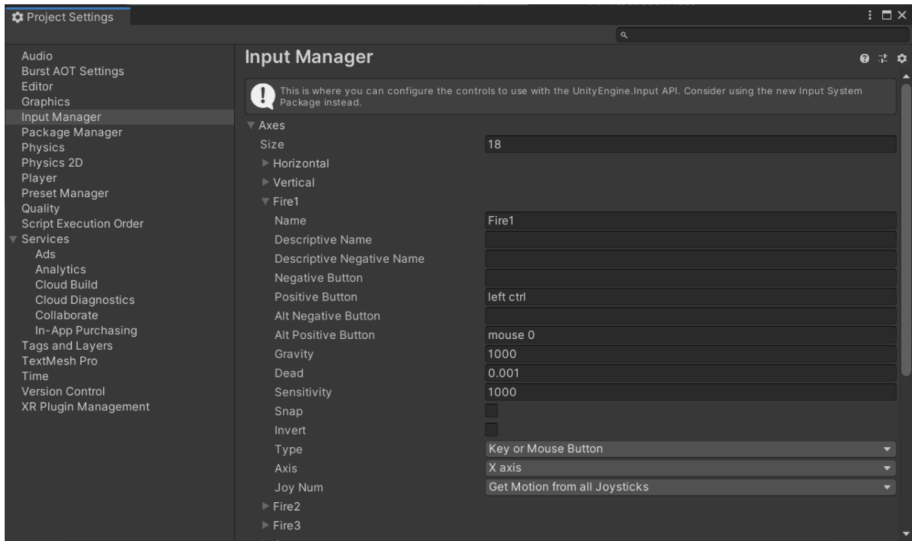

Figure 3.7: Unity input manager

section 2.5. As such, the general guideline towards what part of the user interface is in the graph could divided in:

- Nodes could be either each of the states of the game's user interface, or any element present in any of these states.
- Edges would be a representation of the input required to go from any node to another. Only connections between states or between UI elements would exists, meaning that no connections would define a transition between an UI element and a state, and vice versa.

By persisting this information, in future runs of the tool the planning stage could be avoided by just obtaining the optimal path from the current state to the objective, using, for example, pathfinding algorithms.

3.4. Conclusions

During this chapter, the current tools available were put to the test given the problem that this project is trying to solve. Given the limitations of them, a new tool was designed to be able to solve this issue.

The baseline for the development of the tool has been defined. In doing so, the pipeline of the tool has been divided into 4 modules, one of those already developed. As such, during the development phase, the remaining three modules will be implemented: the scripting module, the decision-making module, and the actors module.

In the following chapter, the development process of the tool will be lay down.

Chapter 4

Tool Development

In this chapter, the development process of the navigation tool will be described, in order to automatically navigate through a user interface using image recognition.

Throughout this chapter, a detailed description of the tool's progression will be outlined. Different subjects, like the decision making, the problems through the development and the architecture will be explained in detail.

4.1. Tool Architecture

After the research phase, the navigation tool started its development. The navigation tool has been implemented in C# as a console application. It is built upon four different modules, that define very different behaviours, and will be expanded upon through this section. In Figure 4.1 a UML module diagram showcases all of the modules that compose the tool, as well as the dependencies between them.

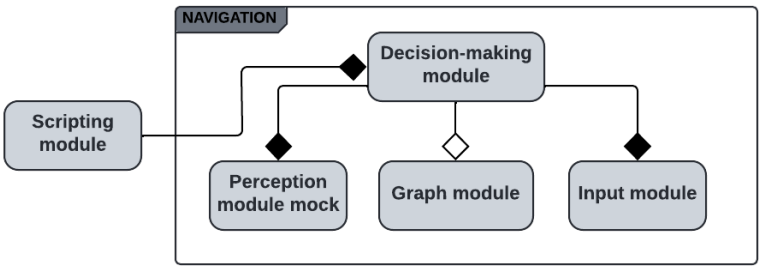

Figure 4.1: Navigation tool UML module diagram

4.1.1. Scripting module

During the first steps of the process, whilst the team was setting up the C# migration of the project, the development of the scripting module was started. This development was divided into two objectives: creating the custom scripting language, and making the main testing loop for the complete tool.

In order to create the scripting language, many options were available, as seen in section 2.4. However, the team decided to select the ANTLR library, not only because of how straightforward it is, but also because it has an readily available NuGet package that could be effortlessly added to the C# Visual Studio solution.

ANTLR requires the definition of rules and tokens on its own custom file. So, to support the commands specified in subsection 4.1.1, a *Grammar.g4* was created, that defines all of the supported commands, as seen in Figure 4.2.

```
grammar CogitoGrammar;

// PARSER //

program: line+ EOF;
line: header | command | end;
header: TEST_NAME ':';
command: instruction SEPARATOR arguments;
end: END;
instruction: GOTO #gotoExpression
 | FIND #findExpression
 | MOVE #moveExpression
 | MODIFY #modifyExpression
 | ASSERT #assertExpression
 ;
arguments: ARGUMENT+;

// LEXER //
fragment LOWERCASE: [a-z];
fragment UPPERCASE: [A-Z];
fragment DIGIT: [0-9];

GOTO: ('G'|'g')('O'|'o')('T'|'t')('O'|'o');
MODIFY: ('M'|'m')('O'|'o')('D'|'d')('I'|'i')('F'|'f')('Y'|'y');
ASSERT: ('A'|'a')('S'|'s')('S'|'s')('E'|'e')('R'|'r')('T'|'t');
END: ('E'|'e')('N'|'n')('D'|'d');

SEPARATOR: ('->');

TEST_NAME: LOWERCASE (LOWERCASE | UPPERCASE)* DIGIT*; // i.e. testCase01
ARGUMENT: '"' ((LOWERCASE | UPPERCASE | ' ')+ | DIGIT+) '"'; // i.e. "settings"

WHITESPACE: [ \t\f\r\n]+ -> skip;
COMMENT: '#' [^\r\n]* -> skip;
```

Figure 4.2: *Grammar.g4* file

ANTLR then generates a series of class files from this *Grammar.g4* file. Although the NuGet package supposedly does this build automatically, there seem to be an issue with the solution that did not allow this build to happen. Thus, a manual build was required every time the grammar is changed.

After the build is done, the following C# classes are created:

- `GrammarLexer` is in charge of the tokenization of the text into lexical rules, using the lexical definitions in the `Grammar.g4` file, that act as the end points of the tree.
- `GrammarParser` uses the text tokenization processed by `GrammarLexer` and generates the parse tree using the rules defined in the file, that will then be run through during the execution.

- `GrammarListener` define methods that act upon entering and exiting each of the nodes of the parse tree.
- `GrammarVisitor` define methods that act upon being in each of the subtrees generated from each of the rules defined in the grammar.

It is important to mentioned that both `Listener` and `Visitor` have a `Base` version (`GrammarBaseListener` & `GrammarBaseVisitor`), intended to be inherited from to create scripting behaviours.

4.1.1.1. Test runner

After the whole build process was done, it was time to create the main testing loop. Thus, a data structure that saves the sequence of commands to be executed in each test was created, in order to then run all the tests at once. These test have to be defined inside a custom scripting file with the *.cogts* extension.

To be able to save and execute every test from these .cogts files, the following classes work in unison to facilitate the desired outcome:

- `Test` works as the data structure that saves every requested command from the scripting file. Once all the commands are properly saved, upon a request to run the test it will run all the saved commands in succession, calling the necessary functions to complete the navigation, and the required asserts to ensure that the test fails or succeeds, showing the results to the user.
- `GramarCustomVisitor` is a subclass of the visitor class `GramarBaseVisitor`. It is in charge of processing the grammar parse tree, visiting every instruction node from the parse tree (i.e., visiting the `goto` command rule) and creating the required `Test` object using the commands and arguments that are present in the file.
- Finally, the `TestingProcess` class contains the main loop of the tool. It reads a series of *.cogts* file and process them using the generated parser and lexer. Then, the parse tree generated will be read thoroughly by the `GramarCustomVisitor` class developed.
  In order to execute these tests optimally, a new thread, hosting a concurrent queue that saves each test created by the visitor, will be in charge of executing them as they come.

4.1.2. Decision-making module

Now that the whole scripting module was developed, there is a requirement to add navigation support to each of the navigation commands. As such, different AI techniques were researched for the purpose of this automation, as seen in section 2.2 and section 2.3, but it was decided that the GOAP planning technique was the most suited for the job, given how it isolates the connection between states, making adjustments much less cumbersome.

In order to implement the navigation planner, different C# libraries were analyzed, and although some were promising, the team landed on using the *mountain_goap* library (Muller, 2022). This library centers around the creation of agents, which isolated the planning process locally. In order to implement any behaviour, *mountain_goap* agents use composition to add to them the necessary goals and actions required to defined the desired behaviours. Plus, its extensive documentation made it the best candidate available.

Once the library is fully integrated into the solution, it was time to develop the navigation behaviour, as explained in subsection 3.3.3. The whole behaviour of this module is defined by the `Planner`, which contains an agent in charge of all of the task planning. Using the composition architecture provided by the *mountain_goap* library, different actions are injected into the class' agent, to be used in the different goals that will be planned.

- During the **preprocessing goal**, the important information required to take movement decision is stored into the agent state variables, this includes:
  - The first action, `LabelingAction`, takes all the labels used in the perception module, like in Figure 4.3, and selects which ones would describe a user interface element that infers a direct connection or an indirect connection to the desired goal, using the current UI elements' assigned labels as the only source of information.

    Going a bit deeper, the label provided by the command is considered the "primary" label, meaning that an element with this label would define a direct connection to the desired state, and then, using the available information inside this label (any important word inside the label, excluding connectors), the planner would select a series of "secondary" labels, which means elements with them assigned have an indirect connection to the desired state, meaning that, by repeatedly going through them, a direct connection may be found in the process. If the label provided by the user is not present in the `planner_labels.json` file, an error sent to the user would interrupt the execution of the whole command.
  - Then, the planner executes the `MainMenuAction`, which will check what the main menu is in the current state, by picking the menu structure with the biggest area covered on the screen.
  - Finally, the last action required, `LocateElementAction`, saves the most promising user interface element, based on the labels saved in the `LabelingAction` step. It can be either found using the "primary" or "secondary" label saved in that action, but if no element is found to have any of them, the user will be notified with an error.
- Afterwards, during the **movement goal**, the `MovementAction` is called, which uses information filtered to select the navigation actions necessary depending on the situation.
  If the UI element selected during the preprocessing is either an input action, or is present in a secondary menu, it will simply selected the element, using the linked key or button in the input action's case. However, if the element is

```
{
    "labels": [
        {
            "id": "accessibility",
            "handlers": [
                "settings / accessibility settings",
                "accessibility",
            ]
        },
        {
            "id": "audio",
            "handlers": [
                "sound settings",
                "volume control",
                "audio",
                "audio & rumble",
                "audio settings"
            ]
        },
        {
            "id": "graphics",
            "handlers": [
                "display and graphics",
                "graphics settings",
                "visual",
                "advanced graphics settings",
                 "basic graphics",
                 "display",
                "advanced graphics",
                "video",
                "display & graphics",
                "graphics configuration",
                "display"
            ]
        }
    ]
}
```

Figure 4.3: `planner_labels.json` file example

present in the main menu, it will make navigation actions to reach the element until it either can not move, or it arrives at the destination.
In any case, if the element chosen has a secondary connection to the desired destination, the planner will loop back to the preprocessing goal to repeat the process again until a primary connection is used.

- Once all the necessary preprocessing and movement goal actions are run, and the planner has arrived at the desired state, a last goal will be planned, the **finish command goal**, which will execute the action specific to each command instruction. For the *goto* command, the `GotoAction` simply selects the element highlighted after all the necessary movement steps are executed. However, for the *modify* command, what the `ModifyAction` will try to do is make every action that usually mean a modification of the field value (that being either checking it when selecting it, or move with left and right to change the current value) until it either arrives the desired value, or all actions are spent and thus the value is not achievable.

To be able to start the whole planning procedure from outside the class, a `RequestCommand` method allows the planner to begin a new command planning, resetting the state variables and specifying what command and arguments have made the request called, so the planner can act accordingly.

4.1.3. Input module

After the whole planner was developed to a working state, the input support for movement actions started its development. The input layers explained in section 3.2 were developed by saving the data on a *input_ layers.json* file, which allows the developer to add to each of the layers (up, down, left, right, select, back, input_action) a series of buttons to use during movement requests, letting the planner try with the different inputs assigned. This structure is devised to allow extensions to add different platforms inputs, by delegating the platform-agnostic work to the input manager.

As the only platform supported right now is computers with Windows as their operating system, an input manager for that platform was developed. It lets different modules request any kind of input supported, being that keyboard or mouse actions. However, during development it was ascertained that DirectX games developed within EA do capture input using their own identifiers. As such, the whole tool had to adapt to that, receiving the key code corresponding to the DirectX representation.

As such, this module consists of the following classes:

- `InputManager` processes the input event requested, utilizing a `KeyEvent` class, that communicates the action (`PressKey`, `MouseClick`, `MouseMove`), key and position of the mouse, if needed. Given the struggles described previously, the KeyEvent supports both Windows and DirectX keyboard and mouse configurations.
- `InputLayers` lets the user adjust the inputs executed upon certain actions, this class reads an `input_layers.json` file, as seen in Figure 4.4. This layer information is then saved in the class, and, via the `RequestInput` method, communicates the input request to the `InputManager` class, selecting the desired element from the layer requested.

```
{
 "pc": {
   "up": [ "DIK_W", "DIK_UP" ],
   "down": [ "DIK_S", "DIK_DOWN" ],
   "left": [ "DIK_A", "DIK_LEFT" ],
   "right": [ "DIK_D", "DIK_RIGHT" ],
   "select": [ "DIK_RETURN" ],
   "back": [ "DIK_ESCAPE" ],
   "input_action": [ "DIK_TAB", "DIK_ESCAPE", "DIK_O" ]
 }
}
```

Figure 4.4: Input layer configuration example

4.1.4. Graph module

Just as the complete pipeline was finalised, and every working module was connected to each other, a request from within the team emerged asking for the creation of a navigation graph, the representation of it abiding to graph theory norms, as explained in section 2.5. Given the inherent nature of any game user interface, it was clear that an digraph was required, since not every action can be mirrored to return to a previous menu.

In order to be able to visualise any graph output, first a visualisation tool was needed. One open-sourced solution to this hurdle is Gephi, which, although it is geared towards network visualization, it can be easily applied to graph visualization as well. Gephi supports a plethora of graph file formats, as seen in Figure 4.5, each of them bringing some advantages and some drawbacks in the feature department.

Figure 4.5: Gephi supported formats and their features

One key feature almost required in order to represent any game's user interface is to be able to be organized hierarchically. As it has been mentioned in section 3.3, any user interface is divided into states, that contain a certain number of elements within them, that then define the environment that the menus are constrained to. As such, a hierarchical representation of these states and elements is required for the navigation tool to work with, which limits the selection of file graph formats to two candidates: GEXF and GraphML.

Although both formats seemed equally useful to the tool, as they both present the necessary features to properly represent the hierarchical structure the user interface is constrained to (as seen in subsection 2.5.1), it was decided to go with GEXF, as it was specifically developed to be used in unison with the Gephi tool.

4.1.4.1. GEXF file formatting

Even during the selection process, where the team was pondering which one of the options was more appropriate for the tool, earlier XML NuGet implementations were researched. Regrettably, all the available options had their shortcomings, either

by not allowing both serialization and deserialization of the file format, or because their API was too obtuse it to be properly used.

Considering these drawbacks, a custom implementation was required to support GEXF file formatting. The C# library `XMLSerialiser` gives an easy to use interface to create serializable classes. However, during development it was made clear that `XMLSerialiser` does not support node nesting, since it considers the nesting as a circular dependency, giving a runtime exception in the process. As such, the GraphML format as well as the node nesting option for GEXF were impossible whilst using this library.

Fortunately, GEXF supports other ways to represent hierarchical structures, so, given the options at hand, parent identifier assignment was the one selected. Thus, state nodes will have this field empty, and element nodes will be assigned their parent state's identifier.

4.1.4.2. Graph representation

With the file formatting problem solved, the graph internal representation behaviour was developed. All of the nodes and edges are added according to movement actions deployed by the planner during the Movement step. Thus, the representation is as follows:

- State nodes will be distinguished by the lack of a **parental identifier** attribute in them.
- Element nodes will both have the **parent identifier** assigned to them, as well as saving the **type** of element they are (either text or icon element).
- Edges are divided into two types to improve visualisation in Gephi:
  - Element to element connection edges, where only the triggering **input** will be stored.
  - State to state connection edges, where both the triggering **input** and the triggering **UI element** (referred to as `bridge`) are stored.

All of the movement done in the planner module that result in either a new element to be highlighted, or a new state to be the current one, will be reflected in this navigation graph by adding the nodes and edges required to represent this connection.

Once all the representation was properly described, the following classes were developed:

- `GexfFormatter` is the class in charge of the serialization and deserialization of GEXF files, in order to persist the graph information across multiple runs. It defines the values of the `GexfMeta` and `GexfGraph` classes, the latter consisting of a set of `GexfAttributes`, `GexfNodes` and `GexfEdges`. All these classes follow the directives that where explained throughout this section.
- The `GraphWrapper` class uses `GexfFormatter` to then act as the internal representation of the navigation graph during runtime. It deserializes the data if

an input file has been provided, and serializes the data to an output when the execution finishes. The methods `AddNode` and `AddEdge` allows external classes to add these two elements to the current representation whenever needed, including attributes, like the `type` and `pid` value for nodes, or the `input` and `bridge` values for edges. These methods make sure that no repeating node or edge is added to the graph data.

4.1.5. Perception module mock

In order to check that all the functionalities are working as intended, the whole pipeline has to be connected. However, given how the current perception module is under development, a way to isolate this behaviour whilst simulating its actions was necessary for the success of the project. Thus, a perception module mock was developed, that emulates a perfect behaviour of the image recognition.

4.1.5.1. Mock development

To be able to emulate this behaviour consistently, several runs of the currently working perception module were done on different screenshots done in different games. After that, the *.json* output file was extracted from each of the screenshots, and any error present in them was fixed, to have what would be a perfect output from the perception module.

Now, with all the data extracted, there is a need to neatly organize it exactly as how the games flow would work. As such, the research done through the Finite-State Machine was applied to this module. Each of the output files will be saved internally in a data structure that persists the output results to the tool. After that, every output saved will then define its connection through code, depending on the input layers defined in the input module explained in subsection 4.1.3. The classes that facilitate this are:

- `StateInfo`, which is used to store the deserialisation of the output .json files from the image recognition module. From these files, it stores the raw UI elements, the raw text, what vertical and/or horizontal menus have been classified, and all the input actions detected in the image processed by the image recognition.
  The UI elements are saved in an internal `DataItem` class, that stores the element type, the obtained contour box and text/icon box, the labels assigned to it, and the value linked to the element (i.e., the volume value).
- `IState` is an interface that defines all of the different `Transitions` methods to be defined in subsequent subclasses that inherit from it (`Down`, `Up Left`, `Right`, `Back`, `Select`, `InputAction`).
  It also acts as a wrapper for the `State` class, that define common behaviours for all states, like the file deserialisation into a `StateInfo` object.
  Each of the states created by inheriting from `State` will then simply define the behaviour from each of the `Transitions` methods.

Once every state was been created and its connections defined, the `Mock` class was created, a singleton acting as the perception module is in charge of presenting the information of the current state, as well as simulating the transitions between states via the input request sent via the `SimulateInput` method.

4.1.5.2. Test cases

Once the module was fully functional, two tests cases were defined in order to tests functionalities across different games, to ensure the tool can work in any game without any tweaks needed.

The test premise follows the one explained in section 3.1. These tests will start from the main menu, reach the settings, arrive at the display section, and then modify the window mode value to "Fullscreen". This flow works in each of the cases as follows:

- In **Battlefield 2024**, the "Settings" menu is directly accessed from the "Main Menu" by pressing the O input action showed on screen. Then, the Display section is selected from the menu bar at the top. Once there, the Window Mode field (named "Fullscreen Mode") is already highlighted, so the value change can then be changed from "Borderless" to "Fullscreen" by tapping to the right.
- In **EA SPORTS™ FC 24**, however, from the "Main Menu" the gear icon has to be selected, then move in the "Customize" menu to "Settings", and in there to "Game Settings". Then, the "Display Configuration" section has to be selected. Once in there, to reach the Window Mode field (named "Display Mode"), a down input has to be made, and then the value can be changed similarly as to how is done in the Battlefield case.

4.2. Conclusion

During this chapter, the architecture of the tool has been showcased, divided into the modules described in the design chapter: the scripting module, the decision-making module, the input module, and the graph module; as well as the isolation of the perception module in a mock state machine, that has allowed us to work without depending on the current progression of the image recognition tool.

In the next chapter, the evaluation of the tool will be tackled.

Chapter 5

Evaluation

This chapter will discuss the current performance of the navigation tool, with a very special focus on time optimization. The navigation module tool should try to minimize the impact on time performance as much as possible.

5.1. Time performance

The current state of the perception module has a computing cost of somewhere between 3 to 4 seconds per image processed, so the objective of this evaluation is to gauge which of the sections of the decision-making module are more time consuming, and obtain the total amount of computing cost added by linking this tool to the perception module.

5.1.1. Methodology

The evaluation has been done monitoring the time employed by each planning step on both the Battlefield 2042 and EA Sports™ FC 24 test cases developed in subsubsection 4.1.5.2. The tests cases will be run 10 times to observe any kind of time variation between runs.

Time has been measured on the following steps: planning stage, preprocessing goal actions, movement goal actions and command execution goal actions. The metric used during this evaluation will be the average time employed by the previously mentioned stages to execute. All of the time step calculations will be surveyed whilst using a machine with an Intel Core i7-8665U CPU and 16 GB of RAM to run the software.

With this information, we can then check both the total time to run these test cases, and add the final time when the navigation tool is fully integrated in the pipeline, taking the approximate time to process each image and adding it to the previous final times.

5.1.2. Results and conclusions

In Figure 5.1 and Figure 5.2 the times for each of the step in each respective test cases evaluated are shown, ordered by when each step was executed. These time tables clearly show that the most time consuming step on average is the planning stage. This is caused by the necessity of the planner to go through all the available options each time a new goal has to be planed for after reaching the previous goal. Also, the first planning stage takes way more time than the rest because of the cold boot.

Aside from that, there is a clear difference between the preprocessing step and the movement step. Whilst the time consumed by the preprocessing step is always low, the movement step has more variability, given how each navigation may require a different amount of navigation actions (sometimes it can be one navigating action, while other times it takes two or more to finish).

Regarding the command finishing actions, the `goto` command takes less time on average than the `modify` command. This is reasonable, given how the `modify` command requires to do certain movements until it arrives at the desired value.

Nevertheless, seeing the average times on Figure 5.3 and Figure 5.4 shows that, even though the planner does take some execution time for itself, the time cost added is almost negligible, specially considering that the perception module would be called more than once throughout the execution of the runs (in these graphs, each call is marked with a green dot in the line graph).

5.2. Conclusion

The main objective of this evaluation was to prove that the navigation tool would not add excessive time cost when linked to the perception module.

After obtaining the results of multiple runs of the software, it was made clear that the amount of time the navigation tool added, although important to consider, was negligible compared to the current time cost of the perception, given how it would add between 0.3 to 1 second extra per perception call.

Phase	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average	St. Dev.
Planning_0	0.469	0.238	0.613	0.230	0.302	0.392	0.442	0.435	0.306	0.532	0.396	0.124
Preprocessing_0	0.014	0.010	0.014	0.013	0.018	0.016	0.016	0.021	0.017	0.017	0.016	0.003
Planning_1	0.038	0.038	0.042	0.039	0.055	0.062	0.060	0.054	0.051	0.044	0.048	0.008
Movement Step_0	0.076	0.067	0.087	0.074	0.072	0.089	0.079	0.069	0.084	0.077	0.077	0.007
Planning_2	0.033	0.055	0.034	0.070	0.064	0.056	0.051	0.055	0.047	0.040	0.051	0.011
Preprocessing_1	0.027	0.024	0.022	0.022	0.032	0.032	0.039	0.025	0.032	0.015	0.027	0.007
Planning_3	0.038	0.045	0.056	0.066	0.074	0.079	0.147	0.075	0.088	0.054	0.072	0.028
Movement Step_1	0.094	0.081	0.093	0.113	0.093	0.108	0.103	0.109	0.084	0.086	0.096	0.011
Planning_4	0.098	0.153	0.147	0.152	0.132	0.114	0.106	0.170	0.200	0.095	0.137	0.031
Preprocessing_2	0.024	0.017	0.015	0.027	0.019	0.022	0.018	0.032	0.029	0.024	0.023	0.006
Planning_5	0.052	0.046	0.097	0.030	0.070	0.045	0.054	0.051	0.143	0.062	0.065	0.032
GoTo Execution_0	0.015	0.024	0.022	0.042	0.024	0.021	0.024	0.022	0.017	0.029	0.024	0.007
Planning_6	0.057	0.058	0.046	0.046	0.056	0.060	0.056	0.056	0.049	0.059	0.054	0.005
Preprocessing_3	0.018	0.015	0.022	0.015	0.023	0.025	0.020	0.021	0.020	0.010	0.019	0.004
Planning_7	0.082	0.035	0.038	0.035	0.078	0.060	0.055	0.059	0.051	0.042	0.054	0.014
Movement Step_2	0.019	0.028	0.028	0.024	0.040	0.026	0.014	0.030	0.017	0.024	0.025	0.007
Planning_8	0.029	0.022	0.021	0.017	0.069	0.044	0.023	0.023	0.021	0.021	0.029	0.016
Modify Execution_0	0.145	0.160	0.143	0.142	0.143	0.166	0.145	0.152	0.144	0.141	0.148	0.008

Figure 5.1: Time table of each planning step applied to Battlefield 2024 test case (in seconds)

Phase	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average	St. Dev.
Planning_0	0.290	0.229	0.572	0.633	0.554	0.456	0.472	0.469	0.432	0.318	0.443	0.118
Preprocessing_0	0.018	0.014	0.025	0.016	0.011	0.010	0.007	0.011	0.015	0.007	0.013	0.005
Planning_1	0.064	0.034	0.073	0.058	0.039	0.041	0.039	0.030	0.046	0.038	0.046	0.013
Movement Step_0	0.083	0.072	0.091	0.088	0.069	0.069	0.071	0.069	0.069	0.067	0.075	0.008
Planning_2	0.125	0.109	0.153	0.169	0.089	0.084	0.065	0.078	0.069	0.177	0.112	0.042
Preprocessing_1	0.023	0.012	0.028	0.029	0.015	0.018	0.010	0.014	0.012	0.026	0.019	0.007
Planning_3	0.043	0.049	0.059	0.124	0.051	0.040	0.026	0.030	0.052	0.074	0.055	0.028
Movement Step_1	0.076	0.083	0.093	0.111	0.066	0.076	0.083	0.077	0.081	0.082	0.083	0.012
Planning_4	0.059	0.079	0.055	0.089	0.050	0.060	0.070	0.055	0.062	0.087	0.067	0.014
Preprocessing_2	0.025	0.076	0.032	0.023	0.043	0.014	0.018	0.013	0.021	0.025	0.029	0.019
Planning_5	0.059	0.176	0.058	0.058	0.055	0.056	0.075	0.039	0.071	0.065	0.071	0.038
Movement Step_2	0.096	0.105	0.083	0.095	0.114	0.085	0.111	0.087	0.075	0.090	0.094	0.013
Planning_6	0.140	0.044	0.061	0.073	0.061	0.066	0.065	0.064	0.037	0.032	0.064	0.014
Preprocessing_3	0.130	0.015	0.026	0.022	0.028	0.026	0.014	0.017	0.018	0.009	0.031	0.007
Planning_7	0.235	0.049	0.065	0.059	0.071	0.063	0.033	0.033	0.047	0.035	0.069	0.015
Movement Step_3	0.113	0.085	0.105	0.091	0.096	0.110	0.078	0.078	0.086	0.078	0.092	0.011
Planning_8	0.031	0.020	0.021	0.024	0.025	0.022	0.015	0.013	0.019	0.020	0.021	0.004
GoTo Execution_0	0.076	0.054	0.060	0.065	0.069	0.068	0.054	0.067	0.065	0.055	0.063	0.006
Planning_9	0.133	0.229	0.102	0.121	0.066	0.084	0.108	0.085	0.093	0.094	0.112	0.045
Preprocessing_4	0.039	0.021	0.020	0.018	0.007	0.020	0.011	0.011	0.016	0.021	0.018	0.005
Planning_10	0.069	0.049	0.046	0.044	0.033	0.036	0.037	0.026	0.037	0.050	0.043	0.008
Movement Step_4	0.027	0.022	0.022	0.018	0.026	0.012	0.012	0.013	0.013	0.010	0.018	0.005
Planning_11	0.028	0.023	0.027	0.030	0.026	0.020	0.018	0.012	0.015	0.017	0.022	0.006
Modify Execution_0	0.091	0.075	0.087	0.101	0.080	0.070	0.067	0.077	0.074	0.064	0.079	0.011

Figure 5.2: Time table of each planning step applied to EA SPORTS™ FC 24 test case (in seconds)

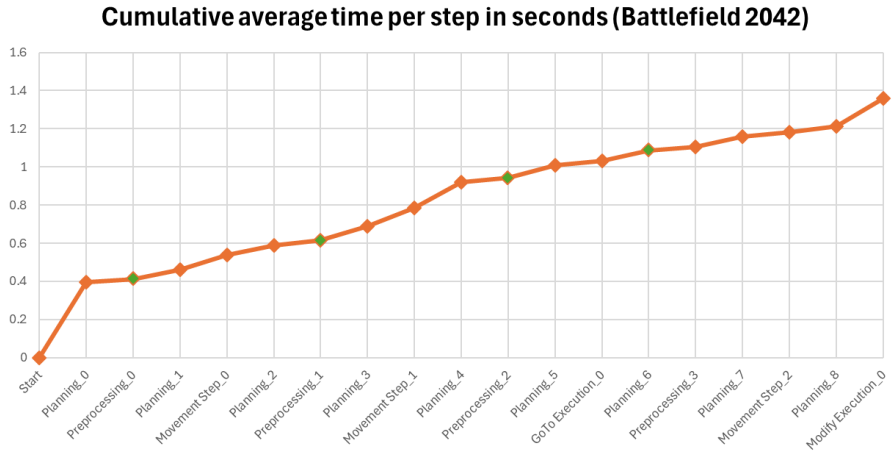

Figure 5.3: Cumulative average time for each planning step applied to Battlefield 2024 test case (in seconds)

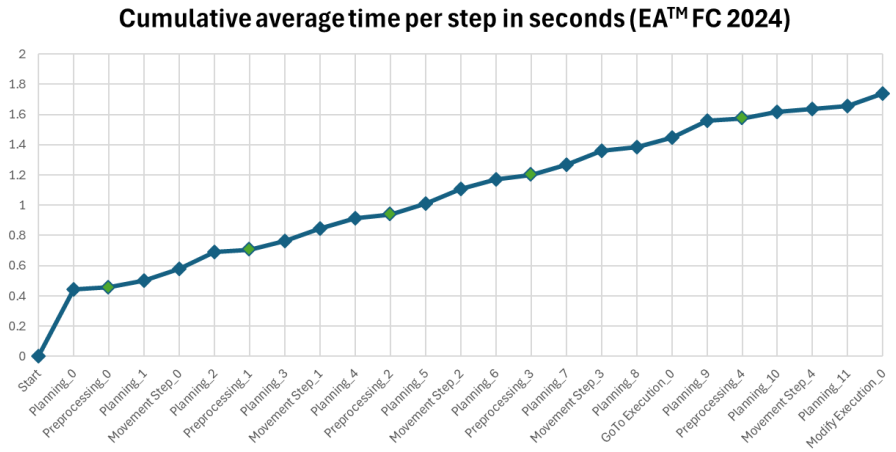

Figure 5.4: Cumulative average time of each planning step applied to EA SPORTS™ FC 24 test case (in seconds)

Chapter 6

Conclusions & future work

"Every moment hesitated is a moment gone from life"
— Nicholas D. Wolfwood

The tedious work that manual testing brings to the table is relevant to many developer teams. These teams should focus their attention to assert more complex fields from the software, and, in game development, from the game. As such, automatic navigation tools are the ideal solution for these teams to streamline development and achieve better results.

This thesis has described the design and development of the navigation tool extension for Cogito. The use of AI task planning techniques in order to identify the correct movement path to arrive at the desired state has made the creation of the tool possible. Besides that, the definition of formal grammars to create a custom scripting language for navigation test has allowed the tool to have a simple interface where all the modules are used. Finally, the creation of a graph representation of the internal structure of the user interface will help in future developments to ease the work of the decision-making of the planner.

The iterative development that this tool has been constrained to has developed the tool from an idea to an actual solution to testers. Each improved version got closer and closer to the finished pipeline that the tool is aiming to be, and will save precious time for developers to improve and fix their user interfaces.

An evaluation of the tool has been conducted, concluding that, although there is a small amount of time added by the task planner to the pipeline, specially during the planning phase, this amount of time was not critical given the current time consumed by the image recognition module.

Future developments and improvements of the navigation tool will be lay down in the next section.

6.1. Future work

Although the tool's current state is more than serviceable, future developments would have the initial idea fully realised. Some of those improvements that could be applied would be:

- Integrate the navigation pipeline with the image recognition module currently in development. This way, the complete pipeline will be realised, and improvements to the tool as a whole can be made easier.
- Currently, although the navigation graph is saved upon test executions, the planner does not use this information to make movement decision through the user interface. This would add another layer of adaptability, given that even developers could add previously produced graphs to make the path selection faster. One problem that could come up is that the user interface changes after an update, but the planner should be able to circumvent this issue by simply just adding a new path to the objective, while detecting that the one selected is deprecated, and marking it accordingly in the saved graph.
- The planner does not take into consideration multiple possible paths towards the end goal. It will always try to commit to only one, and, if that one is incorrect or reaches a dead end, it send a fail message. Ideally, in the future the planner would see any possible path inferred in any state, and try go backwards to return to that state and try all available possibilities.
- The tool is currently only able to be tested on computer machines that run Windows software. One of the extensions to this software that would heavily increase its utility is to support navigation through other systems, such as Xbox Series X, PS5 and Nintendo Switch, so multi-platform developers could easily test in those environments if the user experience is sound. The tool has been built in such a way that extending the input module to allow the test in those platforms would require little developer effort, just requiring the adaptation of the input to abide with every platform's APIs.
- The scripting language currently supports a limited amount of instructions, given the limitations that both the planner and the image recognition have. In the future, more instructions could be developed and supported, such as icon assertions; and improvements to the current ones could be implemented.

Bibliography

Even the smallest person
can change the course of the future

Galadriel, Lord of the Rings

Bastian, M., Heymann, S. and Jacomy, M. Gephi: An open source software for exploring and manipulating networks. 2009a. Avail. at `http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154` (last access, Apr, 2024).

Bastian, M., Jacomy, M., Ramos, E., Girard, P., Yon, G. G. V. and Paterson, D. Gexf. 2009b. Avail. at `https://gexf.net` (last access, Apr, 2024).

Castillo, C. and Quintas, I. *Adaptative tool for automatic test creation of video game interfaces*. 2023.

Dharwadker, A. and Pirzada, S. *Graph Theory*. Proceedings of the Institute of Mathematics. CreateSpace Independent Publishing Platform, 2011. ISBN 9781466254992.

ElectronicArts. Battlefield 2042. 2021. Avail. at `https://www.ea.com/en-gb/games/battlefield/battlefield-2042` (last access, Jan, 2024).

ElectronicArts. F1 22. 2022. Avail. at `https://www.ea.com/en-gb/games/f1/f1-22` (last access, Jan, 2024).

GraphMLTeam. The graphml file format. 2002. Avail. at `http://graphml.graphdrawing.org` (last access, Apr, 2024).

Huggins, J. Selenium. 2004. Avail. at `https://www.selenium.dev/` (last access, February, 2024).

Johnson, M. and Zelenski, J. *Formal Grammars*. 2012.

Keith, C. *Agile game development with SCRUM*. Addison-Wesley, 2015.

Meduna, A. *Formal Languages and Computation: Models and Their Applications*. CRC Press, 2014. ISBN 9781466513457; 1466513454.

Mihov, S. and Schulz, K. U. *Finite-State Techniques: Automata, Transducers and Bimachines*. Cambridge University Press, USA, 2019. ISBN 1108485413.

Muller, C. Mountain goap. 2022. Avail. at `https://github.com/caesuric/mountain-goap` (last access, Apr, 2024).

NetEase. Airtest project. 2014. Avail. at `https://airtest.netease.com` (last access, Jan, 2024).

Orkin, J. *Three States and a Plan: The A.I. of F.E.A.R.*. M.I.T. Media Lab, Cognitive Machines Group, 2006.

Parr, T. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012.

Pichler, R. *Agile Product Management with Scrum: Creating Products that Customers Love*. Addison-Wesley signature series. Addison-Wesley, 2010. ISBN 9780321605788.

Rabin, S. *Game AI Pro 2: Collected Wisdom of Game AI Professionals*. CRC Press, 2015. ISBN 9781482254808.

Ranta, A. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Vallati, M. and Kitchin, D., eds. *Knowledge Engineering Tools and Techniques for AI Planning*. Springer Cham, 2020.

To come to acceptance with things and feelings is rare
and to accept them completely is a miracle.
It's impossible to make that moment come faster by yourself.
Someday it comes unexpectedly.
In order to not become warped or heartless, let it go in a natural way.
Let yourself feel sad when you are,
and let yourself forget when you do.

Adamant
Land of the Lustrous
Haruko Ichikawa