Image processing techniques to extract symbolic features from Atari video games

Moreno Ferrando, Héctor

Curs 2015-2016
Image processing techniques to extract symbolic features from Atari video games

Héctor Moreno Ferrando

TREBALL DE FI DE GRAU
GRAU EN ENGINYERIA EN SISTEMES AUDIOVISUALS
ESCOLA SUPERIOR POLITÈCNICA UPF
2016

THESIS SUPERVISORS
Vicenç Gómez
Gloria Haro
Anders Jonsson
“You think you have a limit. And as soon as you touch this limit, something happens and you suddenly can go a little bit further. With your mind power, your determination, your instinct, and the experience as well, you can fly very high.”

– Ayrton Senna, 1991
Acknowledgements

First and foremost, I would like to express my greatest appreciation to my three thesis supervisors: Vicenç Gómez, Gloria Haro and Anders Jonsson. They have made this project possible by proposing a great idea that captivated me from our very first meeting, and their advice, guidance, and endorsement have been crucial in the development of this project. I would also like to thank UPF professor Ralph Andrzejak for providing insightful guidelines into the making of this report.

I wish to acknowledge the University of Alberta for indirectly providing the resources on which this project is based, in the shape of the Arcade Learning Environment. My work has been possible as a result of their hard work and dedication in the vast field of research that is artificial intelligence.

On a personal note, I would like to thank my family for believing in me throughout the whole process, and also my friends for their help in moments of need. Two of my dearest friends are worthy of a special mention: Àlex, for helping me widen my knowledge on programming; and Lara, for these four fantastic years sharing and learning together.

Last but not least, I would like to deeply thank my partner Eva for helping me keep a positive attitude when difficulties came, and for her unconditional support and encouragement when I needed it the most.
Abstract

Engineers at the University of Alberta have developed a framework called Arcade Learning Environment (ALE), which allows computers to play Atari videogames based on the generation of random agents and a reward system. This software provides, at each time step, information of the game’s RAM, as well as a matrix of pixels that represents the image of the screen. The main goal of this project is to write a program that applies several image processing techniques in order to extract symbolic features from those images. These techniques, implemented in MATLAB, include finding the image’s SIFT keypoints and matching them between consecutive frames, segmenting the frame into connected components and obtaining crucial information about each one of them in order to classify the objects as movable or static, and tracking the objects’ movements around the screen, in order to estimate an interpolation of their trajectory over time.

Resum

Enginyers de la Universitat d’Alberta han desenvolupat un programa anomenat Arcade Learning Environment (ALE), que permet als ordinadors jugar a videojocs Atari basant-se en la generació d’agents aleatoris i un sistema de recompenses. Aquest programa proporciona, a cada instant de temps, informació sobre la RAM del joc, així com una matriu de pixels que representa la imatge de la pantalla. L’objectiu principal d’aquest projecte és escriure un programa que apliqui diverses tècniques de processament d’imatge per tal d’extreure característiques simbòliques d’aquestes imatges. Aquestes tècniques, implementades en MATLAB, inclouen trobar els keypoints SIFT de les imatges i les correspondències entre fotogrames consecutius, segmentar la imatge en components connexes i obtenir informació clau de cada una d’elles per tal de classificar cada objecte segons si és estàtic o mòbil, i rastrejar els seus moviments per la pantalla per tal d’estimar una interpolació de la seva trajectòria al llarg del temps.
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABSTRACT</strong></td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>Index of figures</td>
<td></td>
<td>xi</td>
</tr>
<tr>
<td>Index of tables</td>
<td></td>
<td>xiii</td>
</tr>
<tr>
<td><strong>1 INTRODUCTION</strong></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Context</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1</td>
<td>The author</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2</td>
<td>The Image Processing Research Group</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3</td>
<td>The Artificial Intelligence Research Group</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Objectives</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Structure of the report</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 ON VIDEO GAMES AND THE ATARI 2600</strong></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>What is a video game?</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Brief history of video games</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2</td>
<td>The video game industry</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>The Atari 2600 console</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Survey about video games and the Atari 2600</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Structure of the survey</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Results</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>The Arcade Learning Environment</td>
<td>22</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Compilation of the ALE</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Extracting frames from the games</td>
<td>25</td>
</tr>
<tr>
<td><strong>3 IMAGE PROCESSING TECHNIQUES</strong></td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction to digital image processing</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Scale-Invariant Feature Transform</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Feature matching</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Connected components</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Motion tracking</td>
<td>41</td>
</tr>
<tr>
<td><strong>4 IMPLEMENTING THE ALGORITHM</strong></td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Locating the SIFT keypoints</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Matching the descriptors</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Segmenting the screen into connected components</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Obtaining information about the objects</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Classification according to mobility</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>Following an object’s movements</td>
<td>54</td>
</tr>
<tr>
<td>4.7</td>
<td>Displaying the results</td>
<td>55</td>
</tr>
</tbody>
</table>
Index of figures

Figure 2.1: Lifespan of each generation ........................................ 11
Figure 2.2: Worldwide sales of video game consoles by company ...... 12
Figure 2.3: Total console sales by company ............................... 12
Figure 2.4: Annual worldwide revenue of the industry ................. 13
Figure 2.5: Chain of development of a video game ..................... 14
Figure 2.6: Original Atari logo ................................................... 15
Figure 2.7: Best-selling Atari 2600 games ................................. 15
Figure 2.8: Graphic representation of the survey’s results ............. 20
Figure 2.9: ALE’s execution window .......................................... 24
Figure 2.10: ALE’s recording window ......................................... 25

Figure 3.1: Sampling of a continuous signal ............................... 30
Figure 3.2: Quantization of an image ........................................ 30
Figure 3.3: Detailed pixel composition of a greyscale image ......... 31
Figure 3.4: Composition of a colour image in the RGB space ...... 32
Figure 3.5: Construction of the DoG scale-space ......................... 33
Figure 3.6: Finding the extrema in the DoG scale-space ................. 34
Figure 3.7: Computing a descriptor from the image gradients ....... 35
Figure 3.8: Object recognition through feature matching ............. 36
Figure 3.9: Building a panorama through feature matching .......... 37
Figure 3.10: Example of a graph ............................................... 38
Figure 3.11: Connected component in graph theory ..................... 38
Figure 3.12: Pixel connectivity ................................................ 39
Figure 3.13: Connected components from an image .................... 40
Figure 3.14: Object tracking in surveillance ............................... 41

Figure 4.1: Applying a window to improve feature matching ......... 45
Figure 4.2: Connected components algorithm ............................ 47
Figure 4.3: Different types of bounding boxes ............................ 49
Figure 4.4: Computing the bounding box that surrounds an object .. 50
Figure 4.5: Computing an object’s pixel list ............................... 50
Figure 4.6: Taking the linear indices of a MATLAB matrix .......... 51
Figure 4.7: Bounding boxes on objects that change shape ........... 52
Figure 4.8: Possible outcomes of the tracking algorithm ............... 54
Figure 4.9: Visualisation of the keypoints .................................. 55
Figure 4.10: Visualisation of the matchings ............................... 56
Figure 4.11: Visualisation of the connected components ............... 56
Figure 4.12: Visualisation of the bounding boxes ....................... 57

Figure 5.1: Average keypoints found depending on the threshold ... 60
Figure 5.2: Comparison of the results depending on the threshold . . 62
Figure 5.3: Example of incorrect feature matching .............. 63
Figure 5.4: Example of the proposed feature matching .......... 64
Figure 5.5: Effects of an incorrect window size ................. 65
Figure 5.6: Examples of connected components ................. 70
Figure 5.7: Examples of object classification .................. 72
Figure 5.8: Example of the evolution of three objects’ trajectories . . 73
Figure A2.1: Flowchart of the survey’s questions ............... 98
Figure A5.1: Case of Atlantis .................................. 106
Figure A5.2: Case of Beam Rider .............................. 107
Figure A5.3: Case of Gravitar ................................. 108
Figure A5.4: Case of Phoenix .................................. 109
Figure A5.5: Case of Zaxxon ................................. 111
Index of tables

Table 4.1: Initial parameters of the algorithm .......................... 43
Table 4.2: Input parameters of the showResults function ........ 55

Table 5.1: Average keypoints found depending on the threshold .... 56
Table 5.2: Performance of the SIFT algorithm ......................... 63
Table 5.3: Feature matching comparison ................................. 64

Table A1.1: Worldwide sales by company and console ............. 87
Table A1.2: Annual worldwide revenue of the industry ............. 88
Table A1.3: Best-selling Atari 2600 games ............................ 89
Table A1.4: Games and code-words supported by the ALE ....... 91
1 INTRODUCTION

In this first chapter, we present an overview about all the people involved in the development of this project, and also a brief introduction to the project itself, providing its motivations, as well as its goals and objectives, and the procedure that has been followed throughout the process.

1.1 Context

In this first section, we briefly introduce the author of this project, as well as those people and research groups of the Universitat Pompeu Fabra (UPF) involved in the making and development of its content, and what their roles have been throughout the process.

1.1.1 The author

I am an Audiovisual Systems Engineering student at Universitat Pompeu Fabra in Barcelona. I decided to pursue this Degree because, since a very young age, I have always been hugely interested in everything related to audiovisual content, ranging from music to movies to videogames, from an artistic point of view and, lately, from a more technical point of view as well.

Image and video processing has been an increasing passion of mine since I discovered it in this very Degree. Since then, I have invested a lot of my own time outside the classrooms to widen my knowledge on the subject, learning new and more efficient ways to implement the techniques that I discovered during the subjects of the Degree that cover this area of expertise.

Videogames have always been another interest of mine as well and, although the time I have to play them has decreased year by year, I still consider them to be an enjoyable hobby that allows me to disconnect from my everyday routines for an hour or two and relax while also having fun.

Therefore, when I learnt that there was a chance to carry out a project that combined two of my passions, I knew I had to take it and try to make the most of it. From the very beginning, I really liked the idea of creating a MATLAB program that performed several of these image processing techniques on images extracted from classic video games. I consider myself to be a little bit of a perfectionist, and I do believe this aspect of myself can help me a lot to keep pushing me to achieve not only the best results for this project, but the best possible solutions to the problems that appear along the process as well.
1.1.2 The Image Processing Research Group

The Image Processing Research Group\(^1\) (GPI for its initials in Catalan) of the UPF in Barcelona is a research group that develops algorithms and models to help their studies on image processing and computer vision. Their principal areas of focus range from 3D reconstruction of scenes, to developing tools for video editing, to object identification techniques oriented to logo and brand detection on images and video.

Lecturer and researcher Gloria Haro represents the Image Processing Research Group as a director of this project, guiding me through the process of developing the MATLAB program that applies all the necessary image processing techniques in order to achieve the objectives of the project.

1.1.3 The Artificial Intelligence Research Group

The Artificial Intelligence Research Group\(^2\) (AI) of the UPF in Barcelona is a research group that specializes on the studies of computers’ models of action, inference, and learning. These research areas, all related between them, range from planning, to learning reinforcement, to constraint satisfaction.

Lecturer and researcher Anders Jonsson and post-doctoral researcher Vicenç Gómez represent the Artificial Intelligence Research Group as directors of this project, providing an insight on the information about the games that the group expects to obtain as a result of this project.

1.2 Motivation

One of the recent breakthroughs in AI has been to develop artificial computer programs that reach human-level performance playing Atari video games [1]. The techniques used to achieve this performance are based on deep learning, which has become the state-of-the-art in many machine learning applications. However, despite their impressive performance, deep learning is a brute force learning approach that requires large amounts of data and computational power, and currently fails at solving games that require a long sequential decision steps, like *Montezuma’s Revenge*.

The Atari 2600 video game console was the first home console of global success, selling around 30 million units worldwide during the 15 years in which it was in production. Even now, 24 years after it was discontinued, the Atari 2600 still draws the attention of many

\(^1\)http://gpi.upf.edu
\(^2\)http://ai.upf.edu
enthusiasts and fans of the console, and some of them keep developing compatible video games as their hobby. However, it does not only serve its fans, but also computer scientists have taken advantage of this video game console to help their research on artificial intelligence.

A group of engineers at the University of Alberta have developed the Arcade Learning Environment, a framework that allows scientists to take their research on artificial intelligence (AI) one step further. This framework, which runs on top of Stella (an Atari 2600 emulator), allows the generation of random agents, which can be later used to play the Atari video games. Another advantage of this framework is that it provides the information of the game’s RAM memory at each time step, which allows for a better understanding of the behaviour of the games. There are many reasons behind the choosing of this particular video game console as the base for this research. There are close to a thousand different games available, and Stella can emulate most of them, as well as all the external devices used to control the games (such as the joystick and the buttons, and other peripherals). These games are usually very simple in terms of available movements and actions and, above all, the images displayed on screen are also very simple, therefore several straightforward image processing algorithms can be enough to perform the task. At the same time, though, they present interesting challenges in terms of the artificial intelligence of the agents that appear on the game. But, above all, the main reason for choosing this console is that it was not built for research purposes, but for simple entertainment and, therefore, it allows us to extract real data from it, and not previously prepared information. For these reasons, the engineers at the University of Alberta decided to gravitate their research around this piece of video game history that is the Atari 2600 home console. However, the majority of the existing research has been carried out from a computer science point of view. A motivation for this project is to take an alternative, rather unexplored approach to this research, by analysing what happens on the screen of the games, with the aim of providing crucial information that can help improve the way the Arcade Learning Environment works.

On a different note, but related to this as well, another motivation for this project –and especially for myself– is to be able to make my small contribution to an extremely broad field of research, by giving a different approach which is closer to my area of expertise. In fact, in the Audiovisual Systems Degree we have never addressed the topic of artificial intelligence, so another personal motivation for this project is to learn more about a field from which I have only heard counted times before, while relating it to a field in which I have several years of experience.

1.3 Objectives

As stated in the previous section, research on the field of artificial intelligence has been carried out for years, with a recent breakthrough in the development of artificial programs
with a human-like behaviour when it comes to playing Atari video games [1]. The Artificial Intelligence Research Group of the UPF, in cooperation with the Image Processing Group, are carrying out their research with the aim of learning the symbolic structure of the games. This symbolic structure represents the workflow of the games’ functionalities and outcomes, and it can be taken advantage of through planning algorithms. The present project is an initial step towards this main objective. Here we intend to establish a relationship between the artificial intelligence and digital image processing fields of research, by providing an innovative approach to the current research, performing a study of the screen of the video games through several image processing techniques. Hence, the main goal of this project is to provide an accurate analysis of the screen of a set of Atari video games from a digital image processing point of view, which can be then used in future projects to perform the learning phase through planning algorithms.

More precisely, the goals of the present project are:

- To gain a general knowledge on the history of video games and, more specifically, the Atari 2600 video game console. The overall project proposed by the AI and GPI groups at the UPF is centred on the use of this console, with the help of the Arcade Learning Environment, an essential tool to the process, as it allows us to interact with the games and extract the images of the screen.

- To provide a study of some of the most important and useful digital image processing techniques from a conceptual and mathematical point of view, in order to establish a reference knowledge from which to conceive the bases of the proposed analysis algorithm.

- To adapt these conceptual image processing techniques into the particular case that is the screen of the Atari 2600 video games.

- To implement a MATLAB program that applies these techniques and performs a complete analysis of the screens of the games, finding and matching the interest points on the images, segmenting the objects of the scene, and extracting valuable information about these objects.

- To analyse the results obtained when applying the algorithm, finding the most accurate values for the parameters, and defining what information from the screen can be useful to improve the Arcade Learning Environment. This information will be used in future projects within the scope of the aforementioned research, in order to perform planning algorithms based on the results of our analysis program.

- To set a groundwork and provide the necessary tools for future projects addressing this specific area of research. As stated before, this project is only the first step towards a greater goal within the research being carried out by the AI and GPI groups at the UPF. Therefore, it is highly likely that more projects in a near future are going to cover this topic. Thus, we intend to provide a reference work on which future projects can be based, as well as find the necessary information on how to perform the tasks that lead up to our results.
1.4 Structure of the report

This report is intended to give a detailed insight into all the steps and phases of this project, and explain the procedure that was followed in order to achieve the objectives exposed in section 1.3 above.

In the second chapter, we begin with section 2.1 by giving a brief historical overview about video games in general and the most remarkable milestones that have occurred through the years, in order to put the reader in context, and then, in section 2.2, we take a more detailed look at a specific console: the Atari 2600. This console and its video games are a key aspect about this project, and everything else gravitates around them. In section 2.3, we reflect these concepts with the knowledge of a group of people that have been subjected to a survey on general video game knowledge and, more specifically, about the Atari 2600 console and its games, with the goal of understanding what their knowledge about this topic is, and how it compares between people of different ages. We also introduce the Arcade Learning Environment in section 2.4, another key component of this project, as it allows us to interact with the Atari video games.

Next, in the first section of the third chapter, we introduce the concept of digital image processing, and in the next sections we proceed with the explanation of these techniques from a mathematical point of view, as they shape the majority of this project and, therefore, we intend to set the base for the work performed in the following chapter. The techniques presented are the Scale-Invariant Feature Transform (section 3.2), feature matching (section 3.3), connected components (section 3.4) and motion tracking (section 3.5).

A more computational part of the report follows with the fourth chapter, in which we explain exactly how the process takes place, from the very first idea until the completion of the main MATLAB program and all the scripts and functions created to implement the algorithm. This is followed by—and closely related to—the fifth chapter, which presents an analysis of the results of the algorithm for each step, highlighting its strong points but taking into account its weaknesses and limitations as well. This chapter is focused on three aspects: the effects of the SIFT threshold value, seen in section 5.1, the importance of finding an adequate window size in the feature matching process, explained in section 5.2 and, finally, an overall look at the performance of the algorithm, detailed in section 5.3.

Finally, we wrap up this report in the sixth chapter with a series of conclusions that we extract from the results obtained in the previous chapter, a general overview on the job done and the process of developing the project as a whole, by talking about the things that have been done right, but also the areas in which this project could have been improved, as well as the future work that can be performed from this point onwards, with this project as a base, in sections 6.1 and 6.2 respectively.
2 ON VIDEO GAMES AND THE ATARI 2600

In this chapter, we present a brief historical overview about video games in general, to put the reader in context, and we take a more detailed look at a specific console: the Atari 2600. We also reflect these concepts with the knowledge of a group of people that have been subjected to a survey on general videogame knowledge and, more specifically, about the Atari 2600 console and its games.

We also introduce the Arcade Learning Environment (ALE) – another key part of this project, as it allows us to interact with Atari videogames, as well as the state-of-the-art in the current developments around this software when it comes to interacting with Atari video games and digital image processing.

2.1 What is a video game?

A video game is an electronic entertainment system that takes the inputs from human players and transforms them into audiovisual feedback [2].

The players can provide these inputs through different devices, such as a controller with several buttons, a mouse, a keyboard, or a touch screen. Lately, further advanced technology has been developed, in order to achieve the possibility of interacting with the games with non-electronic inputs, like movements or sounds (e.g. the user’s voice). These inputs are captured by specially designed devices, such as cameras or microphones, and transformed into electronic inputs in order to interact with the game.

A video game is internally programmed to face every possible action that the users present, so when the game receives these inputs, it reacts to the decisions that the players take, and provides a specific audiovisual feedback according to the nature of those decisions. The players’ task, then, is to analyse this feedback and take another decision according to their own criteria, but also based on the games’ instructions, that usually explain an overview of what the possible paths, reactions and outcomes for each particular action inside the video game are.

The visual feedback of the video game is provided to the players through a video device connected to the game – usually a computer monitor or a TV screen –, while the audio feedback is given through the system’s speakers. In the recent years, though, other forms of feedback have been developed, such as vibration on the controller or kinaesthetic communication (i.e. to reproduce the sensations of touch to the users through the device, with movements or vibrations). These news ways to interact with the players are intended to create a more immersive experience, and also provide more information without having to expressly display it onscreen every time.
There are quite a few types of electronic systems – commonly referred to as platforms – available in the market. These platforms go from considerably big boxes with everything integrated within – including the screen itself – to small devices such as smartphones or portable consoles. In the following section, we expose a brief overview on the main aspects and milestones in video game history.

2.1.1 Brief history of video games

The first video game console was built in 1972 [3], but the story of video games began two decades before. In 1948, computer scientist Alan Turing and mathematical economist David Champernowne wrote a computer game that played chess, and they named it Turochamp [4; 5]. This game was never implemented, but it served as a solid base for other computer scientists to start creating the first real video games in history. These video games, though, were never intended to reach the public, as they were only made for research and development purposes. It was in the late 1970s that video games reached the main public, and in a few years they became a huge form of entertainment available to practically everybody. Since then, video games have been evolving constantly, and new technologies have been developed with the aim of improving the systems and games themselves in order to create even more enjoyable experiences to the end-users.

Video game history is often divided into several periods, usually called generations [3], which correspond with the technological breakthroughs that revolutionised the industry of video games and consoles. Here follows a brief overview of every generation, with the most important technological advances and some of the milestones that marked each era.

First generation (1972–1976)

In 1972, Ralph Baer released the first home console in video game history, called Magnavox Odyssey³, while the arcade video game Pong [6] became the first successful video game in gaming rooms. From there, the arcade video game sector escalated very rapidly, as several companies such as Kee Games and Midway established themselves as leading producers of this new, revolutionary forms of entertainment. These first video games used to be stored within the consoles, instead of being provided in separate cartridges. The graphics were very simple, and generally only black and white or combinations of two primary colours. The consoles used basic controllers that consisted of a joystick and a few buttons, and just a few of them disposed of an audio channel [3].

Second generation (1976–1983)

In 1977, the industry experienced its first setback, as sales began dropping and several companies left the industry. However, American company Atari, Inc⁴, released its VCS model, which later would be known as the Atari 2600. With it, the golden age of video games began. Arcade machines became very popular in recreation rooms and shopping

³http://magnavox-odyssey.com
⁴https://atari.com
malls, while home consoles became more affordable and invaded the homes of thousands of players. Single-player games were introduced as a result of the innovation in artificial intelligence, and the introduction of ROM cartridges created the possibility of playing different games on the same console. The graphics were improved, with screen resolutions of 160x192 pixels and up to 16 colours (4 bits), with three audio channels. Some iconic games were released during this generation, such as Taito’s *Space Invaders* (1978), Namco’s *Pac-Man* (1979), Atari’s *Battlezone* (1980), Nintendo’s *Donkey Kong* (1981) and Gottlieb’s *Q*bert (1982) [6].


In 1983, the industry experienced another crash in sales, even bigger than the one in 1977. Only the release of Nintendo’s NES and Sega’s SG-1000 and their popularity in North America helped the industry overcome those difficult times. In this generation, directional pad controllers were introduced, screen resolution increased up to 320x200 pixels, and up to 32 colours were able to be displayed on screen. Some of the most well-known games released during this generation are Atari’s *I, Robot* and *Star Wars* (1983) and Nintendo’s *Super Mario Bros* (1985) and *The Legend of Zelda* (1986) [6].

**Fourth generation (1987–1993)**

This generation supposed the introduction of PC gaming, but the main competition in the market was between Nintendo’s SNES and Sega’s Mega Drive, and their main series of video games, *Super Mario* and *Sonic the Hedgehog* respectively. These new consoles allowed more complex objects to be displayed with up to 4096 colours and with more elaborated movements and actions, which were operated through controllers with up to 8 different buttons. The content of the games began to be more complex as well, with objects represented as 3D models using polygons and shaders. The apparition of the CD-ROM helped store these bigger games, and allowed the introduction of video cut-scenes. In 1990, Nintendo and Atari released the Game Boy and the Lynx respectively, two handheld consoles that would prove to be very successful. Some of the most important games released during this period were Cyan’s *The Manhole* (1987) –which was the first game sold in a CD-ROM–, Squaresoft’s *Final Fantasy* (1990), Capcom’s *Street Fighter* (1991), Midway’s *Mortal Kombat* (1992) and Cyan’s *Myst* (1993), which had great success in the market [6].

**Fifth generation (1993–1998)**

The polygonal graphics introduced in the past generation were improved with the adding of textures, with screens increasing their resolution up to 576i and the possibility to display up to 16,777,216 colours. Controllers were upgraded as well, with the introduction of analog sticks, small joysticks that allowed for faster, more fluid movements of the objects on screen [7]. The storage capacity of the CD-ROM was increased to 650 MB, allowing for bigger games to be developed. The most successful console of this generation was Sony’s PlayStation, which dominated the market for several years. Nintendo and Sega released the Nintendo 64 and the Sega Saturn respectively, but they never came close to the PlayStation. Nintendo, however, dominated the handheld console market with its
upgraded version of the Game Boy, the Game Boy Color. Some of the most iconic video games released during this era were Blizzard’s Warcraft and SEGA’s Daytona USA (1994), Nintendo’s Mario Kart 64 (1997), Sierra Studios’ Half-Life and Rockstar’s Grand Theft Auto (1998) [6].

**Sixth generation (1998–2005)**

Following the great success of the PlayStation, this generation was again dominated by Sony, with their release of the PlayStation 2. Sega’s Dreamcast, Nintendo’s GameCube and Microsoft’s XBOX were the other notable home consoles of this period, but they never came close to the massive success of the PlayStation 2. The major upgrades in this generation came with Sony and Sega’s increase to 128-bit graphics. Everything else was polished and improved, including the storage capacity of the CD-ROM, the analog controllers and the general playability of the games. This generation also supposed the apparition of online gaming, which allowed for the development of massive multiplayer games. In the early 2000s, mobile phone technology was vastly improved, which allowed for the apparition of mobile phone games. Sony also dominated the handheld console market with their release of the PlayStation Portable (also called PSP for its initials). Some of the most remarkable games in this period were Maxi’s The Sims (2000), Atari’s Enter the Matrix and Sony’s massively multiplayer online role-playing game (MMORPG) Star Wars Galaxies (2003), and Bungie’s Halo 2 (2004) [6].

**Seventh generation (2005–2012)**

The seventh generation of video games marked great progress and technological advances in several areas. The market was dominated by Microsoft’s XBOX 360, Sony’s PlayStation 3 and Nintendo’s Wii. Screen resolutions were increased to HD condition, wireless controllers were introduced, in-game graphics were improved to movie-like CGI levels, consoles were able to play full HD movies, and a great breakthrough was made by Nintendo, adding moving sensors to the controllers in order to play the games with the movements of the user as well as the original buttons and joysticks. Microsoft revolutionised the market with the release of the Kinect, a camera that could track the movements of the players and use it to control the game. Nintendo released the Nintendo DS, the first handheld console to allow Wi-Fi connection. Some of the best-selling games of this generation were Nintendo’s Wii Series (2006–2011), Activision’s Call of Duty 4 (2007), Bungie’s Halo 3 and Epic Games’ Gears of War 2 (2008), and Rockstar’s Grand Theft Auto IV (2009) [8].

**Eighth generation (2012–present)**

The eighth generation saw the big releases of Nintendo’s Wii U, Sony’s PlayStation 4 (also called PS4) and Microsoft’s XBOX One. These consoles supposed great achievements in technology, in terms of graphics, screen resolution and playability. Nintendo and Sony also released respective handheld models, the Nintendo 3DS and the PlayStation Vita (or PSVita), which had initial success, but were later overwhelmed by mobile platforms, such as smartphones, tablets and smart TVs. PC gaming was also massively developed, with powerful game engines that consoles were never able to match. This generation also
supposed the apparition of micro-consoles, such as Nvidia’s Shield and Android’s Ouya. Some of the best-selling games of this era are Blizzard’s *Diablo III* (2012), Rockstar’s *Grand Theft Auto V* (2013) and Activision’s *Call of Duty: Advanced Warfare* (2014) and *Call of Duty: Black Ops III* (2015) [8].

Even though a new generation is considered to start once a new breakthrough in technology is achieved, the companies do not necessarily end production of consoles from the past generations. Here follows a graphic representation of a timeline of the lifespan of the consoles on each of the eight generations:

![Lifespan of each generation](http://reddit.com/r/gaming/comments/3ajnot/the_lifespans_of_home_video_game_consoles)

Figure 2.1: Lifespan of each generation. Source: Reddit

In the graphic we see that the third and fourth generations have been the longest in terms of having consoles in production, with 20 and 29 years respectively. It is notable to remark that some consoles of the fourth generation are still being made today, such as SSD’s XaviXPORT and Sega’s Advanced Pico Beena. From the seventh generation, we find Nintendo’s Wii and Sony’s Play Station 3 still in production, even though consoles from the eighth (and current) generation are being prioritized by these companies, and those of the seventh generation will soon stop production. Another point to take into account is that the console that made the second generation prevail with the arrival of the third generation is the one around which this project is centred: the Atari 2600. This console alone stayed in production for 8 years longer than any other console in the second generation, making it one of the first globally successful video game home consoles. With the passing of each generation, video games became more popular amongst the users, and the apparition of home and handheld consoles made video games available to a broader audience. These consoles rapidly took over the market, and the affordable prices of their consoles and games –propitiated by massive production chains– helped them overcome the popularity of the old arcade machines in gaming rooms. In Figure 2.2 below, we present a graphic representation of worldwide sales –expressed in millions of units– of the most successful video game consoles from Nintendo, Sony, Microsoft, Sega and Atari, the five biggest companies in the industry [9].
The most successful consoles of all time are Nintendo’s DS series, with 213 million units sold, followed by the Game Boy series, with 201 million units. Sony’s PlayStation and PlayStation 2 have also done very well in the market, with 102 and 155 million units sold respectively. Microsoft’s best-selling console was the XBOX 360, with 84 million units, Sega’s Genesis sold 31 million units, and the Atari 2600 sold 30 million units worldwide. An aggregate of all the worldwide sales can be seen in Figure 2.3 below. Nintendo and Sony are the dominators of the video game console market, with 49.8% and 33.5% of the sales respectively. Far off, we find Microsoft and Sega, with 8.5% and 5.9% respectively, and Atari is last with only 2.4% of the sales, though Atari stopped production of consoles in 1996 and went through rough times.

---

6 Refer to Appendix 1, section A, for a table with the full data.
2.1.2 The video game industry

Video games need to be produced, developed, promoted, and sold to all the users around the globe. The economic sector in charge of successfully carrying out these operations is known as the interactive entertainment industry—also more commonly referred to as the video game industry [10]. Like every other industry in the economic sector, the video game industry’s main objective is to be profitable through the years, maintain its status and, if possible, increase revenue year by year. A graphic representation of the industry’s annual worldwide revenue[^7] is shown in Figure 2.4 below, from 1992 to 2015.

![Figure 2.4: Annual worldwide revenue of the industry](image)

This chart reflects the worldwide revenue (in billions of US dollars) of the video game industry through the years, from 1992 to 2015. In blue, we present the real values obtained from independent studies carried out every year, while in green we can see the values represented by taking into account the inflation in the US as of 2016 (i.e. converting each year’s revenue value into 2016 US dollars) [11].

As seen in the chart, the video game industry has grown quickly through the years, and nowadays it is able to compete against other leading industries in the entertainment sector, such as the music, film and TV industries. This increase in the importance of video games comes mainly from the development of portable devices that allow to play anywhere, anytime—such as Sony’s PSP and PS Vita—and the introduction of smartphone games (games that can be played in the phone itself, making them accessible to a much broader audience). Another key factor in the vast growth of the industry is the quick development of several Asian markets, which have a massive impact on the annual revenue of the sector, as they englobe large quantities of people.

The industry provides jobs to millions of people around the world, with very different disciplines and areas of expertise within the chain of development of a video game, as shown in Figure 2.5 below.

[^7]: See Appendix 1, section B, for a table with the full data.
At the beginning of the chain of development, we find those in charge of providing the necessary investment of capital in order to fund and promote new games. The games are originated in the minds of the designers and the artists, and then are produced by the developers through several programming and modelling techniques. After that, the games need to be published to the market and distributed not only all over the world, but to online platforms as well. Finally, at the end of the chain –but probably its most important link–, we find the end-users of the games, i.e. the players that, through the platform of their choice, will play the video games [10].

A very important part of any end-user’s gaming experience is the platform used to play the games. In the next section of this chapter, we take a more insightful look at the Atari 2600, one of the most iconic gaming platforms in video game history.

### 2.2 The Atari 2600 console

As seen in section 2.1.1, the video game industry had a slow start, and in the early 1970s video games were only played in gaming rooms, with the popular arcade machines and their simple but addictive games. 1972 saw the arrival of the first video game home console, and several companies saw the potential success in those entertainment systems, so they invested in developing home video game consoles. It was in 1977 that the American company Atari, Inc. released their first big home console, the Atari VCS (short for Video Console System) [12]. A great variety of arcade video games were adapted to the home consoles, and within a few years this platform became a referent in the market. The VCS was only the second console in production (after Fairchild Camera and Instrument’s release of the Channel F in 1976) to feature the use of external cartridges [13], instead of the common built-in games of the other consoles. In the winter season that year, things began to go downhill for the video game industry, and only the VCS was able to overcome the crash in the market. Atari decided to license very popular arcade games, and with their release of Space Invaders in 1978, the golden age of arcade video games began [14]. Sales of the console grew exponentially every year and, in 1982, the Atari VCS received some technical upgrades, and was renamed Atari 2600. In conjunction with third-party companies, Atari developed a great library of video games compatible with the 2600, and thus gained total domination of the market.
In the next years, though, other companies began developing more technologically advanced systems, and the Atari 2600 started to suffer the passing of time. After several price drops and some changes in marketing strategy, the Atari 2600 was finally discontinued in January, 1992, 15 years after its release, making it one of the longest lasting video game consoles of all time [12]. As seen in Figure 2.2, in section 2.1.1, Atari sold around 30 million units of the Atari 2600 worldwide during its run of 15 years, and billions of video games compatible with the console were sold over the course of more than 30 years. In Figure 2.7 below we present a graphic of the all-time best-selling video games of the Atari 2600 home console [8], expressed in millions of units.

As of today, the Atari 2600 is still considered to be the mother of all video game consoles as we know them, and lots of people still play it. Video games for this console are still being made (mostly by fans and enthusiasts), and the console still piques the curiosity of engineers and developers around the world. To give just one example, this project gravitates around the Atari 2600, as well as the work done by engineers at the University of Alberta, Canada, with their development of the Arcade Learning Environment, destined to help research on artificial intelligence and based upon the Atari 2600 as well.

---

8 https://walldevil.com/19973-atari-logos-video-games.html
9 See Appendix 1, section C, for a table with the full data.
2.3 Survey about videogames and the Atari 2600

Nowadays, lots of studies and statistics on the video game industry can be found, either online or on written publications. However, these studies tend to be very broad in terms of specific knowledge concerning this subject, as they are often centred on general aspects about the industry itself, and not so much on the real experience of the players. Therefore, a more concise survey has been carried out for this project, oriented to a diverse group of people, from different genders, ages and occupations. The aim of this survey is to put into perspective the information presented in the previous sections of this chapter, and compare it with the knowledge of this group of people and see how their perception and experience in this field varies according to the different genders, age ranges and occupations. With this, we intend to provide closure to the introductory part of this chapter about video games in general and the Atari 2600 console in particular.

2.3.1 Structure of the survey

The survey consists of several parts, but not all of them are mandatory for everyone that answers its questions; in fact, some of them do only appear according to the answer to some of the previous ones. First of all, we ask for the basic information about the person answering the survey, such as the gender, age, and occupation, as well as whether they play video games or not. According to the answer to this question, they are then redirected to different sections of the survey.

If the answer to this specific question is ‘Yes’, they are then asked how many hours per week they play, on which platform, and their personal opinion on how important they believe video games are in nowadays’ society. If the answer to the question is ‘No’, they are then asked what the reason why they do not play is, and their personal opinion on the importance of video games in society as well. In both sections, though, they are asked whether they know about the Atari 2600 video game console. This question has three possible answers. If the answer is ‘No’, they are directly brought to the last section of the survey. However, if the answer is ‘Yes’ or ‘Maybe’, they are then asked what their knowledge about it is, which of its games they know, and whether they play (or have played) with the console. If the answer is ‘No’, they are brought to the last section of the survey, but if the answer is ‘Yes’, they are then asked what their experience with the console is, and what games they have played.

Finally, in the last section of the survey they are asked to give their opinion on the purpose of this project, their knowledge on image processing, and to rate several image processing techniques on how useful they believe those can be for the outcome of this project.

---

10 Refer to Appendix 2, section A, for a complete list of all the questions in the survey. As explained previously, the survey depends on the answer given to some specific questions. In section B, we present a flowchart that schematizes the possible outcomes of the survey according to the respondent’s answers.
2.3.2 Results

The survey has been answered by a total of 63 people, ranging from classmates, to coworkers, to family members, to professors at the university. Figure 2.8 below shows a series of graphic representations of the results.

**Gender**
- 71.4% Male
- 28.6% Female

**Age**
- 68.3% 20 - 25
- 9.5% 25 - 30
- 6.3% 30 - 35
- 6.3% 35 - 40
- 9.5% > 40

**What is your occupation?**
- 38.1% Work
- 28.6% Study and work
- 33.3% Study

**Do you play video games?**
- 68.3% Yes
- 31.7% No

**How many hours a week do you spend playing videogames?**
- 58.1% 0 - 5
- 23.3% 5 - 10
- 9.3% 10 - 15
- 9.3% > 15

**What platform/s do you use?**
How do you rate the importance of videogames in our daily lives?

```
Answers of those who do play video games

<table>
<thead>
<tr>
<th>Rating</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.3%</td>
</tr>
<tr>
<td>1</td>
<td>11.6%</td>
</tr>
<tr>
<td>2</td>
<td>7%</td>
</tr>
<tr>
<td>3</td>
<td>7%</td>
</tr>
<tr>
<td>4</td>
<td>9.3%</td>
</tr>
<tr>
<td>5</td>
<td>2.3%</td>
</tr>
<tr>
<td>6</td>
<td>20.9%</td>
</tr>
<tr>
<td>7</td>
<td>14%</td>
</tr>
<tr>
<td>8</td>
<td>14%</td>
</tr>
<tr>
<td>9</td>
<td>4.7%</td>
</tr>
<tr>
<td>10</td>
<td>7%</td>
</tr>
</tbody>
</table>
```

```
Answers of those who do not play video games

<table>
<thead>
<tr>
<th>Rating</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20%</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>5%</td>
</tr>
<tr>
<td>3</td>
<td>25%</td>
</tr>
<tr>
<td>4</td>
<td>5%</td>
</tr>
<tr>
<td>5</td>
<td>10%</td>
</tr>
<tr>
<td>6</td>
<td>25%</td>
</tr>
<tr>
<td>7</td>
<td>25%</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

Why don't you play videogames?

```
<table>
<thead>
<tr>
<th>Reason</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No time</td>
<td>41.9%</td>
</tr>
<tr>
<td>Boring</td>
<td>15%</td>
</tr>
<tr>
<td>Not interested</td>
<td>15%</td>
</tr>
<tr>
<td>Don't like them</td>
<td>15%</td>
</tr>
<tr>
<td>Other</td>
<td>15%</td>
</tr>
</tbody>
</table>
```

Do you know the Atari 2600 console?

```
Answers of those who do play video games

- 41.9% Yes
- 27.9% Maybe
- 30.2% No
```

```
Answers of those who do not play video games

- 15% Yes
- 15% Maybe
- 70% No
```
What is your knowledge about this console?

Which of the following Atari 2600 games do you know?

Have you ever played games on this console?

How would you rate your experience with this console?
Which of the following Atari 2600 games have you played?

What do you think of the idea of a computer playing games on its own?

How would you rate your knowledge on image processing?

Rate the following image processing techniques according to how much you believe they could help improve the computer's performance when playing the game

Figure 2.8: Graphic representation of the survey’s results
Of the 63 people that answered, 45 are men, while 18 are women. The majority of those who answered (specifically 68.3%) is between 20 and 25 years old, while the others are homogeneously spread between the ages of 25 and 40. 33.3% of the respondents are currently studying, 38.1% have a job, and 28.6% combine both their studies and their jobs.

Of the total of 63 people that have answered, 68.3% (i.e. 43 people) do play videogames; 58.1% of them spend up to 5 hours playing per week, 23.3% spend between 5 and 10 hours, 9.3% spend between 10 and 15 hours, and another 9.3% spend more than 15 hours playing video games every week. The most used platforms are, in order of appearance, the PC, Sony’s Play Station, the smartphones, and Microsoft’s XBOX. Of these 43 people, the arithmetic mean of their perception of the importance of videogames in our society is 5.4 on a scale from 0 to 10. Of the 43 that play video games, 30 (i.e. 69.8%) have knowledge of the Atari 2600 console.

Of the remaining 20 people that do not play video games, the vast majority state that their reason not to play is because they are not interested, while a few would like to but have no time for it. It is interesting to note that the arithmetic mean of the perception of the importance of video games in our society for those who do not play video games is 3.8 out of 10, almost two points lower than it is for those who do play. There is also a contrast on their knowledge of the Atari 2600: only 30% of those who do not play do know about this platform. Of the 36 people that know the console, 23 have only heard of it, while the rest have had some friend or relative own one, but never owned one themselves. 22.2% (i.e. 8 out of 36) of the ones that know the console have played with it, 2.8% (i.e. 1 out of 36) still do, and the remaining 75% (i.e. 27 out of 36) have never played with it. The most known Atari 2600 games for those who know the console are Ms. Pac-Man, Space Invaders, Tennis and Freeway. Of those 9 people who have played with the console, their rating of their experience with it is 6.6 out of 10, and the most played games are Ms. Pac-Man, Space Invaders and Freeway.

Finally, in the last section of the survey, when asked on their opinion about this project, 36 people (57.1%) have found it to be interesting, 16 (25.4%) believe it is a very good idea, 10 (15.9%) do not understand the purpose of this project, and 1 (1.6%) does not care about it. The arithmetic mean of their knowledge on image processing is 4.3 out of 10. With this knowledge, they rate the importance of background segmentation in this project with a 3.3 out of 5, object detection with a 3.5, object classification with a 3.4, object count with a 2.6, and the tracking of movements with a 3.5 out of 5.

With these results, we can extract some conclusions about the group of people that have answered the survey. Looking at the answers individually, we can clearly see that almost the totality of the men who have answered the survey do play video games, while the women mostly do not. Also, the hours spent playing per week go hand in hand with the occupation of the respondents: those who only study tend to play more than those who have a job, and even more than those who combine both. Another interesting conclusion to extract is the fact that the personal opinion on the importance of video games in our daily
lives is seen substantially differently depending on whether the respondent does play video
games or not. Finally, we have also noted that, even though the respondents do not appear

to have an extensive knowledge on image processing techniques, they understand the
importance of each one of them in the process of implementing the proposed algorithm.

2.4 The Arcade Learning Environment

The Arcade Learning Environment\textsuperscript{11} (commonly referred to as ALE for its initials) is a
framework developed at the University of Alberta, in Canada, that allows developers to
study the behaviour of artificial intelligence, with the main goal of allowing computer
programs to play Atari video games through the generation of random agents and a reward
system [15]. This framework runs on top of Stella\textsuperscript{12}, an Atari 2600 free, multi-platform,
open-source emulator, which is compatible with the majority of Atari games. It can emulate
all of the external devices of the Atari 2600, such as the joystick or the buttons, which
allows for a very accurate reproduction of the behaviour of the video games. There are
several reasons to focus on a specific console like the Atari 2600: a vast collection of over
900 very different games (single-player and multi-player), simple action models, and high
emulation speed provided by Stella, and the possibility to use Stella as a generative model
in order to implement search-based and learning-based methods [16].

In terms of this project, the ALE is a key part of the development phase. A specifically
crucial functionality that the ALE has is that it allows, while playing the game, to extract
the information of its RAM memory at each time step, as well as a matrix of pixels that
represents the image on screen at each point. In this project, we take advantage of the latter,
in order to compile a library of frames of several video games. In order to do so, though,
we first have to compile the ALE into our system. In the next section we briefly explain
what is needed to compile it, to later extract the frames from the games\textsuperscript{13} and thus compile
a library of testing games, on which we are to apply the image processing techniques and
carry out an extensive analysis in order to extract the symbolic features and representative
information about every game.

2.4.1 Compilation of the ALE

In order to compile the ALE\textsuperscript{14}, we need to install the following tools into our system (if
they are not installed already):

\begin{itemize}
\item [13] In order to have a good testing database, we have obtained a library consisting of over 1800 games from
http://www.atarimania.com/rom_collection_archive_atari_2600_roms.html
\item [14] We recommend the use of a Linux-based operating system (such as Ubuntu) for this task.
\end{itemize}
CMake
This set of free, open-source tools is intended to help developers when building and testing new applications or computer programs. In our case, it is useful because it can create a C++ project from a single .cpp file. In order to install CMake, we need to run the following command from the terminal:

> sudo apt-get install cmake

Make
This useful tool can compile and create an executable file from a C++ project by reading its Makefiles—the files that specify a set of instructions that Make has to follow to be able to compile the program. In order to install Make, we need to run the following command from the terminal:

> sudo apt-get install make

SDL library
The Simple DirectMedia Layer (SDL) library provides support to a program when dealing with audio, keyboard, mouse, joystick, and graphics hardware via OpenGL. In order to install the 1.2 release of the SDL Library, we need to run the following command from the terminal:

> sudo apt-get install libSDL1.2-dev

With these tools installed in the system, we need to make sure that we have downloaded the ALE folder from the developers’ website. Then, we just need to access the root folder of the ALE. Once there, we need to create the C++ projects from the .cpp files in the ALE’s folder with the help of CMake. In order to do so, we need to call CMake and specify that we want to use the SDL Library, as well as build the examples. However, we do not need to use the RL-Glue (Reinforced Learning Glue) support interface for this case. Therefore, we need to run the following commands:

User> cd ALE
User/ALE> cmake -DUSE_SDL=ON -DUSE_RLGLUE=OFF -DBUILD_EXAMPLES=ON

When the C++ project is created through CMake, we just need to build it and compile it with Make. We can also set its parameter \( j \) in order to specify the number of operations that can be run simultaneously:

User/ALE> make -j 4

---

15 https://cmake.org
16 https://gnu.org/software/make
17 https://libSDL.org
With this, ALE is now compiled into the system. For example, in order to play a game called Game, stored in the ROM with filename game.bin, we can do so through the command line as well, with the following line:

```
User/ALE> ./ale -game_controller fifo -display_screen true game.bin
```

This call needs to be performed with several input arguments before the path to the filename of the ROM. The main arguments are the following:

- `-game_controller`: selects an ALE interface that defines how Stella communicates with the player agent. By default it is not set, so it is mandatory to set it when performing the call. Its possible values are:
  - `fifo`
  - `fifo_named`
  - `rlglue`

- `-display_screen`: displays (or not) the game screen while the ALE plays the game. By default, it is set to `false`. Its possible values are:
  - `true`
  - `false`

Running this line will begin execution of the ALE, and the game indicated will be played. If we set the `-display_screen` option to be true, then a new window will appear in which the progress of the game is displayed, as shown in the example Figure 2.9 below for the games Ms. Pac-Man (left) and Freeway (right):

![Figure 2.9: ALE’s execution window. Ms. Pac-Man (left) and Freeway (right)](image)

---

18 Note that the path to the ROM file has to be set from within the ALE folder to an outside folder containing the ROM files.

19 Information extracted from the *Arcade Learning Environment Technical Manual (v.0.5.1)*, as well as running the `--help` command from within the ALE itself with the line:

```
User/ALE> ./ale --help
```

For more information on the input arguments to the call, see Appendix 3.
The ALE will play the game until the round ends, either because the system has won the game, or it has been defeated. A key aspect of this project comes from analysing these frames and applying several image processing techniques to the images at each time step, in order to obtain representative information of the game. In the next section we present a simple way of retrieving the game’s frames through the ALE itself.

### 2.4.2 Extracting frames from the games

In the ALE folder downloaded from the project’s website, there is a very useful C++ script called `videoRecordingExample.cpp`, which is one of the scripts that we had to build and compile with the help of CMake and Make in the previous section. This script provides the tools to record the screen of the game at each time step, and store it in a folder within the ALE’s project. To execute the program, we just need to run the following line from the terminal:

```
User/ALE> doc/examples/videoRecordingExample game.bin
```

The program will run the ALE with the ROM file that we introduce as an argument to the call\(^{20}\), and will proceed to create a new folder inside the ALE’s project, called `record`. While the ALE plays the game, it will show its progress through a screen (as explained in the previous section, also shown, for the games *Space Invaders* (left) and *Montezuma’s Revenge* (right), in Figure 2.10 below) and, at each time step, it will store the image of this screen into the folder.

![Figure 2.10: ALE’s recording window. *Space Invaders* (left) and *Montezuma’s Revenge* (right)](image)

\(^{20}\) Note that the ALE will only recognise the ROM if its filename coincides with what the ALE supports. See Appendix 1, section D, for a table with all the supported games in the library, as well as the code word recognized by the ALE associated to each one. The file name of each ROM must be the code word in order for the ALE to recognise the game. The ROMs supported by the current release of the ALE can be found inside the ALE’s folder itself, at `ALE/src/games/supported`. 

An issue that we have noticed with this approach is that, for every call, no matter what ROM we set as an argument, the program always stores the images of the screen into a folder called record. If we run the program and the record folder already exists with captures from another game, the script will overwrite the existing folder and create a new one with the same name, removing all the previous captures as well. This problem has been solved by modifying the .cpp file in order to adapt it to whatever input ROM file we want to record the images from. More specifically, the part where the record folder was created has been changed, and now the program will take into account the name of the ROM, and create a new folder every time it is executed, with the name record_filename. Therefore, the line:

```cpp
std::string recordPath = "record";
```

has been changed to:

```cpp
std::string aux = argv[1];
aux = aux.substr(aux.rfind_last_of("/")+1);
aux = aux.erase(aux.find("."));
std::string recordPath = "record_" + aux;
```

We first create a new auxiliary variable called aux, with the content of the input of the call to the program (i.e. the path to the ROM file). From the full path we take only what comes after the last slash, which will correspond to the filename and extension of the ROM. Then we erase the dot and the extension of the file, and we use this resulting filename in order to make the name of the folder variable according to the input argument of the call to the program. Therefore, the program does now perform its functionality in a more efficient way, creating a new folder named record_filename every time it is called with a different ROM file.

Another minor issue in the process of recording a game’s screen over time is the monotony of this task. Having a library with a considerable number of ROM files, it can take a long time to create a full database of frames of each game. Therefore, a simple bash script called recordGames.sh has been developed in order to automatize this process so that it can be carried out in the background. This bash script is implemented as follows:

```
#!/bin/bash
# Execute the videoRecordingExample.cpp for each game in the ROMs folder
for filename in ../ROMs/*.bin; do
    timeout 10 ./doc/examples/videoRecordingExample "$filename"
done
```

This for loop goes over each file in the ROMs folder, and for every one of them, it executes the videoRecordingExample.cpp program. This execution is programmed to last exactly 10 seconds for each game, by using the timeout function. The idea is to have exactly the same number of frames for each game, although the different frame rates of several games
influence in the final result. The advantage of having added this slight modification to the videoRecordingExample.cpp program does come to play now, because the program itself will create a different folder for every iteration of the loop. In order to run this bash script, we first need to concede all permissions of use, and then simply execute the script and let it perform its task, with the following lines:

User/ALE> chmod 755 recordGames.sh
User/ALE> ./recordGames.sh

With this, we have generated a library of 10 seconds worth of screenshots (somewhere around a thousand images, depending on the frame rate of each game) from 57 different games\textsuperscript{21}, which we will use in the following chapters in order to carry out the main tasks that shape this project.

\textsuperscript{21} This release of the ALE supports 62 different games. Of the initial library of over 1800 ROMs, we are now left with 57 different games, as 5 of the supported games were not included in the library.
3 IMAGE PROCESSING TECHNIQUES

This project intends to provide an alternative approach to the study of artificial intelligence, applying several image processing techniques to the frames of Atari video games extracted using the Arcade Learning Environment. In this chapter, we introduce the concept of digital image processing, as well as some of the most important techniques that will be of use to us when developing the program. In order to implement these techniques, we must first study them and their mathematical background.

3.1 Introduction to digital image processing

Within the broad field of scientific research that is digital signal processing, we define digital image processing as the branch that specializes in the analysis of digital images through computer algorithms, which apply different mathematic principles in order to obtain and –if needed– modify the information provided by those images [17]. The advantage of focusing on digital images is that some of the downsides of the original analogical images (such as noise, blur or imperfections) can be avoided or, at least, partially corrected beforehand. Conceptually, an image can be mathematically interpreted as a two-dimensional function \( f \) of a pair of spatial coordinates \((x, y)\) [18]:

\[
(x, y) \mapsto f(x, y)
\]

The values of \( f \) at each point \((x, y)\) of the plane, determined by the scene they are representing (also referred to as source), must be positive and finite. These values depend on the illumination present on the scene \( i(x, y) \) and the individual reflectance characteristics of each of the objects that are displayed \( r(x, y) \). Therefore, the function for a specific point can be expressed as the combination of these two values [18]:

\[
f(x, y) = i(x, y)r(x, y)
\]

Where

\[
0 < i(x, y) < \infty \quad \text{and} \quad 0 < r(x, y) < 1
\]

This is the continuous, theoretical definition of a digital image, from a mathematical point of view. However, a computer is not able to represent continuous information. Therefore, in order to convert this continuous function into a digital, two-dimensional signal, the processes of sampling and quantization must be carried out.

Sampling

In signal processing, the process of sampling consists in converting a continuous function into a discrete signal by taking samples of the values of the function at equally spaced points in time [19], defined by the sampling rate, as can be seen in Figure 3.1 below.
In the particular case of digital images, then, the process of sampling is used to create a matrix of size $M \times N$ of finite positions or samples, where $M$ is the size on the $X$ axis, and $N$ is the size on the $Y$ axis.

**Quantization**

The process of quantization consists on dividing the range of continuous values of a function into equally separated intervals, and assigning a discrete, integer value to each step [20]. In the particular case of image processing, this process consists on dividing the outputs of the continuous function into $L$ different grey levels. Given a number of bits, defined as a positive integer $k$, the number of grey levels is defined as $L = 2^k$ [18]. Most commonly, digital images are encoded using 8 bits, resulting in a total of 256 grey levels, which means that these images they take discrete integer values ranging from 0 (pure black) to 255 (pure white). In Figure 3.2 below we show an example of this case of $k = 8$ bits, together with three additional examples, using the values of 1, 3 and 5 bits.

With this, we can define a digital image as a matrix of size $M \times N$ in which, for each pixel, a discrete quantization value derived from the continuous function $f$ is assigned [18], as can be seen in this general definition and in the example in Figure 3.3 below.

$$I = \begin{pmatrix}
    f_{0,0} & f_{0,1} & \cdots & f_{0,N-1} \\
    f_{0,0} & f_{1,1} & \cdots & f_{1,N-1} \\
    \vdots & \vdots & \ddots & \vdots \\
    f_{M-1,0} & f_{M-1,1} & \cdots & f_{M-1,N-1}
\end{pmatrix}$$
Greyscale images are represented through one single channel. However, a special case to take into account is the case of colour images. These images are more complex, and require more channels in order to be represented accurately [21]. There are several ways of representing them digitally, through different colour spaces such as the RGB (which uses a combination of red, green and blue to represent the images) or the CMYK (which uses the primary colours cyan, magenta and yellow, and also black). In this project, we use the RGB channel to represent the colour images. In the example in Figure 3.4 below, we show a representation of how a colour image is stored in a computer.
3.2 Scale-Invariant Feature Transform

One of the most well-known and powerful computer vision algorithms is the Scale-Invariant Feature Transform (also referred to as SIFT for its initials), proposed by David G. Lowe in 2004 [22]. This algorithm provides a set of image descriptors that allow for object or scene recognition through reliable matching techniques. These descriptors are invariant to translation, rotation and scaling, and robust to several image transformations, such as noise, blur and changes on the point of view [23]. In broad terms, the algorithm finds a set of points of interest (also called keypoints) in the images, and then extracts a set of descriptors for each keypoint, which are robust and unique to each point of interest. Therefore, given two different images of the same scene with some differences between them, and the set of keypoints and their respective descriptors, a highly reliable matching can be performed between the two. Next are the four steps that shape the algorithm [22].
Detection of scale-space extrema
The first step in the process is to find the candidates to be identified as a keypoint in the images. In order to do so, a series of optimized cascade filters are applied to the images, therefore reducing the computational cost of the procedure, as the most extensive operations are only applied to the resulting candidate points. These candidates are detected in the scale-space of the images, which is defined as the convolution between a variable scale Gaussian function $G$ and the original image $I$:

$$ L(x, y, \sigma) = G(x, y, \sigma) \ast I(x, y) $$

Where

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

This detection is made more efficient by convolving the extrema found in the Difference-of-Gaussian (DoG) function with the original image. Its efficiency comes from the simplicity of its calculation, as well as the stability of the extrema it finds. The Difference-of-Gaussian, then, is computed as the difference between two consecutive scales, separated by a constant scalar $k$, as shown in the graphic representation in Figure 3.5 below. Once all the DoG have been computed, the image is downsampled and blurred, and the process is carried out again. Therefore:

$$ D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) \ast I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma) $$

Figure 3.5: Construction of the DoG scale-space. Source: [22]
Once the DoG scale-space is built, the extrema (either maxima or minima) are detected progressively by analysing each point and contrasting it with its eight neighbours in the current scale, as well as its nine neighbours in the previous scale and the nine in the next one. This point will be selected as a candidate to be a keypoint only if it is the biggest (maximum) or smallest (minimum) of them all.

This technique is shown in Figure 3.6 below. The studied point is painted in orange, while all the other blue points are the ones to which the current point will be compared. With this, all the potential SIFT keypoint candidates are computed.

Figure 3.6: Finding the extrema in the DoG scale-space. Source: [22]

**Keypoint detection**

After computing all the candidates to be interest points, the algorithm rejects those candidates that are unstable or that do not provide reliable information about the image (such as points on the edges of the objects, which could be the cause of unreliable matchings). The authors have found that the quadratic Taylor expansion [24] is the most optimal function to find the interpolated location of the extrema. This function is applied to the DoG scale-space function with an offset $\mathbf{x} = (x, y, \sigma)^T$, in order to centre it at each candidate point:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x}$$

Let $\hat{\mathbf{x}}$ be the location of the current maximum (or minimum). This location is defined by taking the derivative of the Taylor expansion and setting it to zero. Then, in order to discard those candidates that are unstable or unreliable, the function value at that specific extremum is applied:

$$\hat{\mathbf{x}} = - \frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2} \frac{\partial D}{\partial \mathbf{x}} \rightarrow D(\hat{\mathbf{x}}) = D + \frac{1}{2} \frac{\partial D^T}{\partial \mathbf{x}} \hat{\mathbf{x}}$$

All those points with a value of $D(\hat{\mathbf{x}})$ smaller than 0.03 are rejected, and so all the definitive keypoints are now detected.
Orientation assignment

Once all the keypoints have been established by the algorithm, the next step is to assign a consistent orientation to each point of interest. These orientations are invariant to image rotations, because they are computed taking into account the image properties around the interest point. As seen in the first step, the keypoint candidates are found in the scale-space \( L(x, y, \sigma) \). The smoothed image \( L \) with the closest scale (thus, making the result scale-invariant) is used in order to compute the magnitude \( m(x, y) \) and the orientation \( \theta(x, y) \) of every keypoint as follows:

\[
m(x, y) = \sqrt{(L(x + 1, y) - L(x - 1, y))^2 + (L(x, y + 1) - L(x, y - 1))^2}
\]

\[
\theta(x, y) = \tan^{-1} \frac{L(x, y + 1) - L(x, y - 1)}{L(x + 1, y) - L(x - 1, y)}
\]

The algorithm then proceeds to compute a histogram of all the gradient orientations in a neighbourhood around each keypoint, and then selects the peaks (which correspond to the dominant orientations) and assigns them to the respective keypoint.

Keypoint descriptor

The final step of the process consists on assigning a unique descriptor to each keypoint, which now consists of a centre position \((x, y)\), a specific scale \(\sigma\), and a dominant orientation \(\theta\) \([23]\). In order to do so, the algorithm computes the magnitude and orientation of all the points in a neighbourhood around every keypoint location, as seen in the previous step. Then, with the help of a Gaussian weighted window, the dominant orientation is found, together with its magnitude. In the example in Figure 3.7 below, the image gradients are shown in the left, with the Gaussian window represented by the blue circle. In the right, the histograms of magnitudes and orientations are shown.

![Image Gradients and Keypoint Descriptor](image.png)

Figure 3.7: Computing a descriptor from the image gradients. Source: [22]

This image shows an example with a 2×2 array of histograms, but the authors have found that the most optimal results are achieved with a 4×4 array of histograms, with 8 orientations each. Therefore, the final descriptor computed for each interest point is a 128-element array \((4 \times 4 \times 8 = 128)\) \([22]\), with the results of the orientation histograms and their respective magnitudes.
With these four steps, the SIFT algorithm computes very robust keypoints on the images, that allow for reliable feature matching, another very powerful image processing technique explained in the next section of this chapter.

### 3.3 Feature matching

Feature matching is a digital image processing technique strongly related to the finding of image descriptors and points of interest with methods such as SIFT, explained in the previous section [25]. For a specific keypoint from a certain image inside a sequence of frames from the same scene, feature matching consists on finding a keypoint from another image of the set whose Euclidean distance to the point is the smallest. The Euclidean distance of a set of samples is computed by summing the squared differences of each sample, giving a notion of the distance that separates them in the space [26]. Having the smallest Euclidean distance allows us to define this currently studied point as the original keypoint’s nearest neighbour (NN). However, this condition is not enough to mark a match as definitive. The method in [25] proposes that a match will be reliable only if the ratio between the distances from the original point to its nearest and second nearest neighbours is smaller or equal than 0.8:

\[
\frac{d(k, NN)}{d(k, 2NN)} \leq 0.8
\]

If this condition is fulfilled, that matching is considered to be reliable and, therefore, it is added to the list of matchings. Matching algorithms are very useful in object or scene recognition problems, because they provide the necessary tools to match two or more images from a specific scene but with some differences between them (such as changes in point of view, rotation, etc.). A common practice is to have a reference image, with a specific object or logo, and a set of images of one or more scenes in which the reference object or logo appears (not necessarily in all of them). SIFT keypoints are computed on the reference image, as well as in every image in the sequence, and then they are matched. With this, we can perform reliable object recognition, as seen in Figure 3.8 below.

![Figure 3.8: Object recognition through feature matching. Source: [27]](image-url)
Another application of feature matching is to build panoramic images from two different points of view of the same scene (a). This can be achieved by finding the interest points on both images (b) and matching their descriptors (c). Then, a simple homography can be applied in order to merge both images into one, taking into account the matches specified between the keypoints (d). An example of this process can be seen in Figure 3.9 below.

Figure 3.9: Building a panorama through feature matching. Source: [26]
3.4 Connected components

In order to define the concept of connected components, we must first define the basic concepts of graph and subgraph.

Let \( V \) be a set of vertices and \( E \) a set 2-element subsets of \( V \), known as edges. In basic mathematics and computer science, a graph \( G \) is defined as a pair of sets \( G = (V, E) \) that satisfies \( E \subseteq [V]^2 \) [28]. An example of a graph with 28 vertices and 24 edges is shown in Figure 3.10 below.

![Figure 3.10: Example of a graph](image)

A subgraph \( G' \) is a graph formed from a subset of several vertices and edges from the original graph \( G \) [29]. A connected component is a subgraph \( G' \) all of whose vertices are connected to each other but disconnected from all the rest, and therefore can only be accessed from other vertices within the same subgraph [30]. Following the example shown in Figure 3.10 above, a graphic representation of the 5 connected components that form this graph is shown in Figure 3.11 below.

![Figure 3.11: Connected components in graph theory](image)

Graph theory can be easily interpolated to the case study of image processing by considering every pixel as a vertex of a graph which, in turn, corresponds to the entire
digital image. Therefore, the concept of connected component can be applied to image processing as well. In order to define this concept, though, we must first define two preliminary concepts.

**Pixel connectivity**

Let \( p \) be the pixel with coordinates \((i,j)\) from a specific image. Pixel connectivity refers to which pixels will be taken into consideration when searching for connected components [31]. There are two possible cases, as shown in Figure 3.12 below, in which \( p \) corresponds to the red pixel, and the orange pixels are the ones taken into consideration when searching for connected components:

- **4-Connectivity**: only the pixels directly above, below, left and right of the pixel that is currently being studied, \( p \), are taken into consideration (image (a) in Figure 3.12 below). Therefore, and given the pixel \( p \), the set of pixels that the algorithm is going to analyse is:

\[
N_4(p) = \{(i-1,j), (i,j-1), (i,j+1), (i+1,j)\}
\]

- **8-Connectivity**: in addition to the pixels directly above, below, left and right of the current pixel \( p \), in this case also the top left, top right, bottom left and bottom right pixels are taken into consideration (image (b) in Figure 3.12). Therefore, and given the pixel \( p \), the set of pixels that the algorithm is going to analyse can be expressed as the union of the previous set with the one described in this section:

\[
N_8(p) = N_4(p) \cup \{(i-1,j-1), (i-1,j+1), (i+1,j-1), (i+1,j+1)\}
\]

![Figure 3.12: Pixel connectivity. 4-Connectivity (a) and 8-Connectivity (b)](image)

**Pixel value**

As explained in section 3.1, a digital image is stored in a computer as a matrix of pixels. Each pixel has a position \((i,j)\), which corresponds to its row and column within the matrix, and a value that represents its brightness. In very simple, binary images, these values can be 0 (for white pixels) and 1 (for black pixels). In more complex images, these values can
have a bigger range, as seen in the example in Figure 3.2 previously, for the cases of 2 (a), 8 (b), 32 (c) and 256 values (d). For colour images, the idea is the same, as we can first convert the image to grayscale, and then apply the same reasoning.

A connected component is a set of pixels that fulfil two conditions: all the pixels must be connected between them according to one of the options of pixel connectivity, and also share the same pixel value \([32; 33]\). A good implementation of the algorithm will go over the image matrix (a) and it will assign a label (i.e. colour) to each pixel (b), according to the connected component to which they belong, as seen in Figure 3.13 below (and \(\emptyset\) for the pixels in the background). In the next chapter we take a deeper look to this algorithm.

![Figure 3.13: Connected components from an image. Original image (a) and labelling (b)](image)

In this image, we find several examples of special cases to take into account when applying the connected components algorithm:

- At the top of the original image from Figure 3.13 (a) we can see two objects that are in contact with each other. Their far-right and far-left pixels respectively are adjacent, but their pixel values are different. Therefore, the connected components algorithm will define these two objects as two separate components, and therefore label them with two different labels.

- In the image, we see that there are two separate objects that share the same yellow colour, but they have no adjacent pixels between them. Therefore, the connected components algorithm will classify them as two separate objects as well, with a different label for each one.

- The object at the bottom left corner of the image has a pixel that is not adjacent, but has the same pixel value as the rest of the object. In this case, it has been included
as part of the same object because, using 8-Connectivity, it fulfils both conditions to be part of the same connected component.

In this project, we need to identify the objects in the scene and segment them from the background. Therefore, we will need to implement an algorithm that, given the sequence of frames of a game as input, searches for all the connected components in the images, which represent the objects on the scene.

### 3.5 Motion tracking

The final digital image processing technique covered in this third chapter is motion tracking. Once an object in an image has been detected and identified as mobile, a very useful procedure is to track its movements along a sequence of images. Any object in an image is considered to be mobile if, along several frames on the sequence, its position varies with respect to the background of the scene. Then, the object’s position is tracked at each frame, with the aim of building a position histogram or a trajectory chart [34]. This technique can be used in a number of applications, such as video surveillance, as seen in an example in Figure 3.14 below, by analysing the frames and detecting suspicious movements from someone.

![Figure 3.14: Object tracking in surveillance. Source: Nils T Siebel](http://www.siebel-research.de/people_tracking/)

---

22 [http://www.siebel-research.de/people_tracking/](http://www.siebel-research.de/people_tracking/)
There are several potential problems with this procedure, such as occlusions in the scene, sudden movements of the camera, changes in viewpoint, or the presence of several instances of the same object. There are several algorithms that perform reliable object tracking, such as Mean-Shift [35], taking into account all of these potential issues that may occur in the process. However, in this project we take a more simple approach. Once the object is identified and found in several frames of a sequence, the position coordinates of its centroid are stored, in order to build a trajectory diagram that allows us to understand its movements.

With this, we conclude this chapter that serves as an introduction to several, very powerful and useful digital image processing techniques. In the next chapter we commence to dissect the implementation of the MATLAB program that applies these techniques with the purpose of extracting crucial information from the screen of several Atari video games in the library compiled in section 2.4 of the previous chapter.
4 IMPLEMENTING THE ALGORITHM

As stated in the introductory chapter of this report, the main goal of this project is to extract symbolic features from the Atari video games’ screens through several digital image processing techniques, with the aim of providing enough information to the Arcade Learning Environment to improve its performance. After the conceptual introduction to some of the most important and useful of these techniques in the previous chapter, we now present a further explanation on the implementation of the algorithm that makes for the body of this project.

The algorithm starts by locating the interest points on the image and their respective descriptors, and then matching them between consecutive frames. Then, the screen is segmented into connected components in order to identify the objects, and pixel information about each object is retrieved, which helps classify them according to their mobility. It also provides information about the evolution of an object’s position, with the aim of identifying patterns between objects that behave similarly. Finally, the algorithm provides four different visualisations of the results obtained during the process.

To carry it out, a MATLAB script with the initial definition of the input parameters has been developed, along with some calls to several functions that perform the adequate image processing techniques seen in the previous chapter, in order to obtain representative information about the screen of the Atari video games. Table 4.1 shows a brief explanation of the three initial input parameters of the main script, together with their data types and default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>int</td>
<td>100</td>
<td>Number of frames that the program skips when gathering the test images.</td>
</tr>
<tr>
<td>th</td>
<td>float</td>
<td>0.025</td>
<td>SIFT threshold.</td>
</tr>
<tr>
<td>win</td>
<td>int</td>
<td>5</td>
<td>Size of the matching window.</td>
</tr>
</tbody>
</table>

Table 4.1: Initial parameters of the algorithm

After the definition of these three initial input parameters, the script proceeds to call to several functions implemented to perform the different image processing techniques that shape the process. The steps followed for this implementation are thoroughly documented in the next seven sections of this chapter.
4.1 Locating the SIFT keypoints

The first step of the process is to find all the SIFT keypoints (and corresponding descriptors) in every frame of the sequence. The code provided by [22] includes an implementation of the SIFT method through the sift function. Therefore, we have defined a function called applySIFT (with path, step and th as input parameters) that adapts the call to this function to the purpose of this project. The process followed by this function is rather simple. First of all, it creates a loop that takes images from the game’s folder (skipping some of them according to the value of step). For every image in this sequence, the function stores its colour and greyscale versions, and then applies the sift function to the greyscale images, as this is defined by the sift function itself, in order to find all its keypoints. As a result of this process, the applySIFT function returns the output parameters imgRGB (a cell array containing all the colour frames from the sequence), img (a cell array with all the same images but in greyscale), keypoints (a cell array with all the keypoints found in every frame) and desc (a cell array with the respective descriptors to each keypoint in every frame).

4.2 Matching the descriptors

With all the keypoints detected, the next step is to match them between consecutive frames. The code provided by [22] includes a feature matching function called siftmatch but, after some trials, we have noted that its functionality is not very adapted to our case of study. Therefore, a new function called findMatches has been implemented, which performs a more reliable matching between keypoints of consecutive frames. This function takes as input parameters the cell arrays keypoints and desc, which contain, respectively, all the keypoints and descriptors from all the images in the sequence. Moreover, a simple but effective window restriction has been applied to the feature matching technique explained previously in section 3.2, established by win, the third input parameter to the function, which is defined at the beginning of the script as an initial parameter (see Table 4.1 above). For every keypoint in a frame at time t, and assuming an object’s movements to be small between this current frame and the next frame at time t+1, we use a simple square window that restricts the program to search for matching keypoints in the image at time t+1 only in an area of the image defined by the size of the window, and centred at the exact location of the keypoint currently being analysed in the image at time t. In the example in Figure 4.1 below, we show a frame at time t (a), with an object and one keypoint, marked with a yellow cross, and the next frame at time t+1 (b), with the same object – whose position has changed by a small amount –, and one keypoint marked in green. In this frame, we also display the position of the window (in orange), which is a square of size 2*win×2*win, centred at the exact position of the keypoint in the previous frame (marked in yellow).

23 For a better understanding of how this function works, see Appendix 4, section A, for the pseudocode.
Once the window is applied, the program is restricted to those keypoints that fall within the window’s region. As explained in section 3.3 of the previous chapter, the matches are established with those keypoints whose Euclidean distance to the original keypoint is the smallest with respect to the others. Let $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ be two points in the plane; the Euclidean distance between them is defined as the length of the segment $\overline{pq}$ that connects the two points [36]:

$$d(p, q) = d(q, p) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Therefore, the program implements this definition between the current keypoint at time $t$ and all the keypoints at time $t+1$ that fall within the window. The keypoint at time $t+1$ with the smallest descriptor distance to the currently analysed keypoint in the frame at time $t$ is selected as a match, and the pair is stored in the output parameter `matches`, which is a cell array that contains all the pairs of matches between consecutive frames in the sequence of frames. A special case to be considered is that of an object disappearing. This means that from one frame to another the keypoints corresponding to that object disappear as well, so the program has to be prepared for this to happen at some point. This has been prevented by adding a simple condition that only if a match for a keypoint is found, both are stored as a pair, but never a single keypoint is to be stored without a pair, because this would produce an execution error in the program. The overall functionality of the feature matching technique is both improved and optimized with the `findMatches` function that we have described, because now, instead of going over the whole frame, it assumes a small movement and minimizes the area of search and, therefore, the computational cost is reduced drastically. This improvement is detailed in the next chapter, when covering the results of the process.
4.3 Segmenting the screen into connected components

As explained in section 3.4 of the previous chapter, segmenting an image into connected components allows us to find all the objects in the scene. The next step of the implementation of the analysis program is to find all the connected components of every frame, in order to be able to differentiate them from the background and, thus, perform a first step towards object classification.

A connected component in image processing is defined as a group of pixels of an image, all of whose components fulfil two conditions: they are adjacent to each other and they share the same pixel value [33]. In the main script, we have included a function called `findObjects`, which is intended to find all the objects in a scene and get crucial information about them. In this section, we cover the first part, while the second part is detailed in the next.

In order to apply this algorithm, we have implemented a new function called `findConnComp` within the `findObjects` function. Its aim is to find all the connected components on the input image. To do this, we start by showing on screen the first frame of the game, and we select a point on the background of the image, in order to get its pixel value. Then, we create a blank matrix with the same size of the input image, and we start analysing pixel by pixel for connected components. This function uses 8-connectivity (see section 3.4 in the previous chapter), because most of the objects in Atari games have shapes that 4-connectivity would not interpret as part of the same object. Therefore, the function goes pixel by pixel, looking at its 8 neighbours, and marking it as visited once done\(^{24}\). If the pixel corresponds to the background, the function assigns a 0, and for every other object uses a different label (see Figure 3.13 in the previous chapter). Every time a new object is detected, the function increases the value of the label by 1, so from top-left to bottom-right, every object has a different label for each pixel that shapes it. In order to optimize the process, the program avoids looking at those pixels that have already been marked as visited, as well as those pixels that fall outside the image boundaries and those that belong to the background.

However, there are numerous objects in different Atari video games which have more than one colour, but it is more unlikely that two different objects will be in contact. For this reason, we have decided to omit the condition of sharing the same pixel value between adjacent pixels, in order for the function to detect those objects that have more than one colour as well. As a result, the `findConnComp` function returns a labelled matrix called `CC`, of the same size as the original image matrix, with 0 in all the pixels that correspond to the background, and a scalar number for each different object. Therefore, and following the example set in Figure 3.13 in the previous chapter, in Figure 4.1 below we show the result of applying our algorithm (b) to that particular image and object disposition (a).

\(^{24}\) For a better understanding of how this function works, see Appendix 4, section B, for the pseudocode.
Figure 4.2: Connected components algorithm. Original image (a) and labelled matrix (b)

It is important to remark that, in this case, after the modification that we have applied to its functionality, the algorithm interprets that the image has only 4 objects, whereas in theory it should indicate that there are 5. With this, we have adapted the algorithm to recognise multi-coloured objects. In the next section, we go one step further and show how to obtain relevant information about every object in each frame in the sequence of images that form the video game.

4.4 Obtaining information about the objects

After segmenting the images into connected components and finding all the objects on the scene, the next step of the program is to get crucial information about every object in particular. This functionality has been implemented within the findObjects function presented in the previous section. Therefore, inside the same loop that goes through all the images in the sequence, and after finding all the connected components (stored in the cell array CC) through the findConn Comp function, the main function calls a very useful built-in MATLAB function called regionprops. This function takes as input parameter the labelled image with all the connected components, CC, as well as a series of options that define exactly what information we want to obtain from the image, and returns a cell array called info, which contains all the information about every object in each frame stored as a structure. In our case, we have limited the call of this function to five options that are to be of returned as fields of the structure for every object in each frame. These five options are described below.

Area
This field of the structure provides a scalar value of exactly how many pixels form that particular connected component. Therefore, for every object in each frame, this field of the structure looks like this:

\[ \text{Area: N} \]

Where \( N \) is the scalar value that corresponds to the object’s area.

Centroid
In geometry, the centroid—or centre of mass—of a closed, non-self-intersecting polygon is defined as the average position of every one of its \( n \) vertices [37]. If we describe the shape as a set of \( n \) 2D points such as \( \{(x_i, y_i)\}_{i=0}^{n-1} \subset \mathbb{R}^2 \), which represent the vertices of the polygon ordered counter clockwise, its total area \( A \) is computed as follows [38]:

\[
A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)
\]

Given its area, the centroid of a polygon is defined as:

\[
\frac{1}{6A} \left( \sum_{i=0}^{n-1} (x_i + x_{i+1}) (x_i y_{i+1} - x_{i+1} y_i), \sum_{i=0}^{n-1} (y_i + y_{i+1}) (x_i y_{i+1} - x_{i+1} y_i) \right) \in \mathbb{R}^2
\]

Or, in a more computational way:

\[
C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1}) (x_i y_{i+1} - x_{i+1} y_i)
\]
\[
C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1}) (x_i y_{i+1} - x_{i+1} y_i)
\]

Where the centroid can be expressed as \( C = (C_x, C_y) \). Therefore, for every object in every frame, this field of the structure will be returned as follows:

\[
\text{Centroid: [Cx Cy]}
\]

Bounding box
In geometry, a bounding box is the smallest rectangle that can enclose a specific set of points, shapes or objects inside. Bounding boxes can be axis-aligned, meaning that the rectangle’s sides will always be parallel to the scene’s axes, or arbitrarily oriented, meaning
that the scene’s axes are not taken into account, and the smallest box possible is computed, no matter what its orientation is [39], as can be seen in the example below:

![Figure 4.3: Different types of bounding boxes. Axis-aligned (a) and arbitrarily oriented (b)](image)

In this image, we show a representation of the same scene with two shapes—a square and a circle—, with an axis-aligned bounding box (a) and an arbitrarily oriented bounding box (b). Note that the sizes of the boxes are not the same, and depending on how the objects are distributed around the screen, one type of bounding box will be smaller than the other. For instance, in this specific example, we can see that the arbitrarily oriented box is quite smaller than the axis-aligned box.

In this project, we will be using axis-aligned bounding boxes for the objects on the games, because the regionprops function returns this field of the structure, for every object on each frame, as a 1x4 array of double values, as follows:

$$\text{BoundingBox: } [x_{ulc} \ y_{ulc} \ x_{length} \ y_{length}]$$

Where $x_{ulc}$ and $y_{ulc}$ are, respectively, the pixel coordinates on the $X$ and $Y$ axes of the rectangle’s upper left corner, and $x_{length}$ and $y_{length}$ specify, respectively, the length of the box along the $X$ and $Y$ axes. In the images in Figure 4.4 below an example of how the array’s values are computed and returned as a field of the main structure.

In the image, schematic representations of one frame of a game (a) and the axis-aligned bounding box that surrounds the object in the scene (b) are shown. These frames can be interpreted as 10x10 matrices, which results in a total area of 100 pixels. Therefore, in this case the object has a total area of 32 pixels, while the smallest bounding box that can enclose the object has a total area of 42 pixels, starting from pixel (3,2) and with 6 pixels of width and 7 pixels of height. Then, the values on this field of the structure for this case should be $\text{BoundingBox: } [3.0000 \ 2.0000 \ 6.0000 \ 7.0000]$.
Pixel list
At each frame, and for every object, this field of the structure contains a list of the pixel coordinates of the exact pixels that form the specific objects. The result is returned as an Nx2 matrix of double values, as follows:

\[
\text{PixelList: } [x_1 \ y_1] \\
[x_2 \ y_2] \\
[x_3 \ y_3] \\
\vdots \\
[x_N \ y_N]
\]

By default, the matrix is ordered in an ascending order of the elements on the first column.
A practical example of how a pixel list is computed is shown in the image in Figure 4.5 below, as well as the resulting pixel list for this particular case:
This figure shows a practical example of the way a pixel list is computed. We can see a frame of a game with an object that has been identified through the connected components algorithm. The pixel list of pairs of coordinates is then built by going over the object from left to right and from top to bottom.

**Pixel ID list**
Apart from the pixel coordinates, the structure also provides another field that corresponds to the linear indices of the matrix [40]. By definition, when a matrix is created in MATLAB, it is also stored as a matrix of linear indices. This means that, instead of having each position of the matrix stored as a pair of coordinates, the values are stored as single indices, following a top to bottom, left to right pattern:

![Pixel ID list example](image)

**Figure 4.6: Taking the linear indices of a MATLAB matrix**

This figure shows an example of the way a pixel ID list is computed when a MATLAB matrix is defined. In this case, we have a 5x5 matrix. Therefore, and following the top to bottom, left to right order, the pixel ID list will be [1 2 3 4 5 6 ... 24 25]. Following with the example frame used on Figure 4.6 above, a pixel ID list of the pixels that shape that particular object in that specific frame is also provided. Therefore, instead of having an Nx2 matrix of pixel coordinates, this time we have an Nx1 array of double linear indices that shape that object, where N is the scalar value that corresponds to the object’s number of pixels.

```
PixelIdxList: i1
           i2
           i3
     ...   
iN
```

With this, we have gathered all the necessary information about every object in each frame of the sequence. In the next section we give an insight on how this information can be used in a simple classification algorithm.
4.5 Classification according to mobility

After obtaining all the pixel information from every object on each frame of the sequence and storing it in a cell array called info, the next step in the analysis program is to establish a basic classification of these objects, in order to determine which are static and which move around the scene. Therefore, a simple classification algorithm according to an object’s mobility has been developed.

The first idea that comes to mind when facing the problem of classifying objects between moving and static is to look at how the size of the bounding boxes that surround them evolves over time. However, several situations may cause problems: in certain games, objects can change shape, be broken, or even disappear, at some point in time. In the sequence of images in Figure 4.7 below, an example of this scenario is shown.

![Figure 4.7: Bounding boxes on objects that change shape. Frames at time t (top) and t+1 (bottom)](image)
In the case of this example, we have a scene with a single object. At the time instant $t$, this object has a size of 35 pixels (a), which results in a bounding box with an area of 49 pixels (b). In the next frame, at the time instant $t+1$, as a result of some action that happened in the game (for example, a projectile hit the object and broke it), the object sees its size reduced to 27 pixels (c). Consequently, the size of the bounding box that surrounds the object has now been reduced to 36 pixels (d). It is obvious that the object remains the same, and that it has not moved, so it should be considered a static object. However, if we took into account only the position of the bounding box, we would end up labelling this object as mobile, because the coordinates of the four vertices of the bounding box have changed.

Another way to check for static objects is to track the position of its centroid throughout the whole sequence of frames. However, we would encounter a similar problem as described with the bounding boxes, because when a static object has its size reduced, its centroid may change slightly of coordinates, but enough for the program to consider that the object has moved.

The classification algorithm that has been implemented for this project takes a different approach to those proposed in the previous paragraphs. The idea is that, knowing exactly which pixels of the image correspond to every object, an object that is static will normally preserve the majority of those pixels intact throughout the whole sequence of frames.

For this, a new function called classifyObjects has been implemented\textsuperscript{26}. The function takes as input the cell array $\text{info}$ and, on each frame, it looks at every object’s field on the structure within $\text{info}$ that corresponds to that particular frame. Then, it checks for pixel coincidences between the current object and all the others in the other frames, and it keeps the one with the most correspondences. In order to avoid the case of a mobile object casually landing in the position of the current object, and to take into account the case of a static object that is broken or changes shape, we have imposed a condition to be fulfilled for an object to be considered static. When the object with most pixel intersections has been identified, the program checks if the ratio between the number of matches and the area of the objects is greater than 90%. If it is, it assigns the object as static. If there are not many coincidences, or the area of coincidence is too small, the object is marked as mobile.

With this, we have implemented a simple but effective classification algorithm for all the objects in the scene, differentiating them between mobile and static. In section 4.7 later, we show how this object classification is applied into the display of the results obtained by the analysis program.

\textsuperscript{26} For a better understanding of how this function works, see Appendix 4, section C, for the pseudocode.
4.6 Following an object’s movements

In addition to classifying all the objects in the screen between mobile and static, an interesting approach is to analyse how they actually move around the scene. The main idea is to track the position of each object’s centroid over time, so that we can see how fast or slow it moves from frame to frame. Then, by analysing the movements of several objects at the same time, we can easily identify patterns and similarities between different objects which, in turn, can give a first idea that they share the same type. Therefore, the result of this algorithm should be a 2D line chart that represents an increasing function in which, whenever the object moves—whatever its direction is—the function grows according to how fast the position of the object changes and, if the object stays still or disappears, the line is shown completely horizontal, to represent no movement. The general idea of the output of this algorithm is shown in the example in Figure 4.8 below.

![Figure 4.8: Possible outcomes of the tracking algorithm](image)

In the example from the image, we show five different scenarios, displayed in different colours for a better understanding. First, in red, we can see a constant, slow movement between two frames, and then, in orange, a slower, isolated movement. After that, in yellow, a very fast movement, followed, in green, by no movement at all and, finally, in blue, the object disappears from the screen, so it cannot be tracked. As seen in section 4.4, one of the fields of the structures within the cell array info corresponds to the centroid coordinates of the object, which consist on a 2-element array (for the X and Y coordinates). Therefore, in order to interpret these coordinates over time, we compute the displacement from the centroid in the frame at time t and the centroid in the frame at time t+1. Then, we find the cumulative sum of this displacement, so that whenever there is no movement, the line stays horizontal, and finally we plot the evolution on a linear, 2D chart. For this, we have implemented a simple function called trackMotion that, given the cell array info with the pixel information of every frame, and a series of numbers corresponding to the objects that we want to analyse, computes and plots a simple 2D line that shows how each of the input objects move over time.
4.7 Displaying the results

The final step of the analysis process is to display the results. For this, a function called `showResults` has been implemented, with the following input parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>show</td>
<td>Display option, to choose between Keypoints, Matches, Connected Components or Bounding Boxes.</td>
</tr>
<tr>
<td>lines</td>
<td>Option to display (1) or not (0) the lines between the matched keypoints.</td>
</tr>
<tr>
<td>sec</td>
<td>Seconds the program will take between displaying one frame and the next.</td>
</tr>
</tbody>
</table>

Table 4.2: Input parameters of the `showResults` function

The function allows for four possible ways of representing the results – specified by the parameter `show` – and, depending on the input option, the function will execute a specific part of the code, determined by a `switch` expression within the function. These four options are thoroughly explained below.

**Keypoints**

This option shows the original frame in the left, and the same frame (in greyscale to improve visualisation) in the right, with its respective keypoints marked with yellow crosses on top of the image.

![Figure 4.9: Visualisation of the keypoints (Berzerk)](image)

**Matchings**

With this option, the original frame is displayed in the left (at time t), and the frames at times t and t+1 are vertically displayed in the right. On top of them, all the matched keypoints are displayed with different colours (yellow, magenta, cyan, red, green, blue and white) and markers (+, o, *, x, s, d, ^, v, <, >). Each keypoint at the time t+1 is painted with the same combination of colour and marker as its match at time t. There is also the option available to print lines of the same colour that match the points between both frames, established by the input parameter `lines`, detailed in Table 4.2 previously.
Connected Components
With this option, the original frame is displayed in the left, and the same frame is displayed in the right, with every connected component (thus, every object) painted in a different colour, given by the colormap parameter of the imshow function. For this case, the parula map is used, which ranges from yellow to blue.

Bounding Boxes
With this final option, the original frame is displayed on the left, and the same frame is displayed on the right (in greyscale, to improve visualisation) with a bounding box enclosing each detected object. Depending on the category of the objects—as found previously in section 4.5—the bounding boxes are painted in green for mobile objects, while those surrounding static objects are painted in red. An example can be seen in Figure 4.12 below.

---

With this, the implementation of the analysis algorithm is completed. The next step of the process, covered in the following chapter, is to test the obtained results and see how different values for different parameters affect the results.
5 ANALYSING THE RESULTS

Once the program is implemented, the next step is to test it and analyse its performance. In this chapter, we present the results of this process of testing the program, accompanied by several tables, graphics and images destined to support these results. We start by focusing our attention into two specific parts of the program introduced in the previous chapter: the effects that the different values of the SIFT threshold have in the total number of keypoints detected in a frame—as well as their accuracy—, and the consequences of varying the size of the window in the matching algorithm. After this analysis, we evaluate the overall performance of the program in different games of the library composed in the second chapter and, all throughout the process, we face the problems and issues that have appeared both along the process and in the analysis of the results.

5.1 Effects of the SIFT threshold

The first part of the program that we analyse in depth is the effects that the values of the threshold have in the resulting number of keypoints detected in an image, as well as the accuracy with which they are located. According to the code provided by [22], the threshold is an optional parameter which defines that all of the maxima of the DoG scale-space below its value are to be ignored. Its values must be greater or equal than zero, and the default value proposed by the authors is 0.01. Logic tells us that the lower the value, the more keypoints the function can find, as more features from the images are to be accepted, so the values of the threshold usually gravitate around 0.03, and greater values will significantly reduce the number of keypoints located by the algorithm. Therefore, as explained in the previous chapter, we have implemented a function called applySIFT, and within this function we call the sift function provided by [22] with a variable called \( \text{th} \) that represents the value of the threshold.

In order to study the effect that the threshold value has in the number of keypoints found by the sift function, we have developed a simple MATLAB script that runs the applySIFT function for a sequence of five frames in six different games, with a variable value for the threshold input parameter. Values greater than 0.045 provide few keypoints for the images, as some of the important features of these images are neglected due to this restriction. In some cases (such as Space Invaders), for a threshold value of 0.07, the algorithm does not find any keypoints. Therefore, for this experiment we have opted for a set of threshold values ranging from 0 to 0.045 in intervals of 0.005. The number of keypoints between frames of the same sequence can be different due to, for instance, a change in the number of objects present in the scene. Therefore, we have limited this test to analysing the average number of keypoints found in the whole sequence of five frames for every game, rather than looking at each frame in particular. The results of this experiment are shown in Table 5.1 below, accompanied by a linear graphic to support this information.
We observe that the smaller the value of the threshold, the bigger the number of keypoints the algorithm finds in the images. For all the games, no matter what the global number of keypoints is (which may depend on the number of objects on screen, for instance), there is a clearly decreasing tendency on the local number according to the different threshold values. This, as said before, is the logic explanation, because the lower the threshold, the more features are taken into account and the more depth the algorithm takes into finding the extrema in the DoG scale-space [22].

### 5.1.1 Finding the adequate threshold value

On paper, this data seems to provide good enough reasons to choose a threshold value of 0 and, therefore, take all the possible features into account, thus getting a greater number of keypoints and, as a result, performing a better analysis of the images. However, more keypoints do not necessarily result in a more accurate analysis of the screen. In order to
prove this statement, we provide a visual example of the results presented in Table 5.1 above. In particular, this example shows the same frame (301) of the game *Space Invaders*\textsuperscript{28}, for the ten different threshold values, and with all the keypoints displayed as yellow crosses on top of each image.

\textsuperscript{28} Refer to Appendix 5 for the visual examples of the other five games.
Looking at these images in Figure 5.2, choosing an accurate threshold value for the program does not seem as obvious as it seemed on paper. We can clearly see that, in the first five cases (i.e. threshold values from 0 to 0.020), the algorithm finds a greater number of keypoints on the images, but we also observe how an important amount of those keypoints are located in less discriminative areas and, thus, they are not useful (for instance, some of them appear on the background, which is a flat surface without any important features from which to extract information). On the other hand, in the last five cases (i.e. threshold values from 0.025 to 0.045) the program finds a fewer amount of keypoints – and some objects are left without any –, but the ones that are found are always located in places of interest of the image. This is due to the fact that all the maxima in the DoG scale-space below these threshold values are ignored, and therefore some features from some objects are not found by the sift function.

Therefore, we must find the right balance between the amount of keypoints the algorithm finds and how representative of the image they are, by combining the data shown in Table 5.1 with what we observe in the images in Figure 5.2 above and the other examples shown in Appendix 5. Considering these situations, 0.025 has been determined to be a good default threshold value for this project, as it provides the better balance between the amount of keypoints that are found and the accuracy of their location in the image, which is crucial for the correct performance of the program.
5.2 Window size in matching

Once all the keypoints in a sequence of frames have been correctly located, the next step in the program is to match them between consecutive images. As stated in section 4.2 in the previous chapter, the code from [22] provides a feature matching function called \texttt{siftmatch}, but its performance in the context of this project is not very accurate due to the many self-similarities that these type of images present. In Figure 5.3 below we show an example of incorrect feature matching performed by the \texttt{siftmatch} function, and displayed with the \texttt{plotmatches} function, also provided in the code from [22].

![Figure 5.3: Example of incorrect feature matching](image)

We use the game \textit{Space Invaders} for this example as well, as it provides a very clear visualisation of the problem of this function. In this game, we have a scene with 36 aliens (i.e. enemies), disposed in six rows with six aliens each. All the aliens on a row have exactly the same shape and attributes, and they are different from all the other aliens. The \texttt{siftmatch} function is limited to finding matches between keypoint descriptors, but does not take into account the spatial information. Thus, and for each row, the function matches all six aliens on that row in the first frame to the first alien on the corresponding row in the next frame, due to the fact that all the aliens have the same shape and attributes, so their keypoint and descriptor information is approximately the same, and the function matches each alien with the first similar one it finds, which is usually the first on each row.
For this reason, as explained in section 4.2 on the previous chapter, a new function called `findMatches` has been implemented. This function is intended to improve the problems shown in Figure 5.3 above, by adding a window restriction to the process of matching. Thus, when going through each keypoint’s descriptor in an image at time $t$, the function only looks for matching descriptors in the image at time $t+1$ in a small area around the position of the keypoint at time $t$. With this, we omit the possibility of matching the keypoints of an object with the ones of another similar object but in a completely different position in the scene.

Another aspect that we have improved is the display of the matches provided by the `plotmatches` function from [22]. As seen in Figure 5.3, all the keypoints are matched by green lines, which makes for quite a difficult visualisation of the results. In this project we propose a revised way of displaying the matches, intended to improve visualisation by giving each pair of matching points a specific colour and marker. We also include the option to display lines between them if wanted. These two cases can be seen in Figure 5.4 below, using the same frames of the game than in the example with the original function in Figure 5.3 above.

![Figure 5.4: Example of the proposed feature matching](image)

Observing these examples, we can clearly see that the feature matching algorithm has been improved consistently, as now the keypoints are better matched between consecutive frames. We almost no longer see keypoints from one object be matched to the keypoints of a different object with the same shape. There are some exceptions in this example, but these are due to the fact that we have chosen to use a big step value between frames, because...
from one frame to the next, the movements are sometimes very small or even inexistent. With a bigger step between consecutive frames, we ensure to have a substantial change in the position of the objects that move, and thus providing the needed scenario for an accurate visualisation of the results. This situation raises the question of which value is the most accurate for the window size. In the examples of Figure 5.5 below we show two extreme cases that can result in incorrect feature matching.

Figure 5.5: Effects of an incorrect window size. Examples of too small (top) and too big (bottom).

On the top images we see two frames of a scene at times $t$ (a) and $t+1$ (b). In the scene there are two objects, with two keypoints detected on each one. We focus our attention in the keypoint marked in yellow in (a), and our aim is to match it to the corresponding keypoint in (b), in which the objects have slightly changed their position. For this, we set a small window in (b), centred at the exact position of the yellow keypoint in (a), and we look for all the keypoints within the window area. The problem in this case is that, being
the window so small, the objects have moved enough for their keypoints to fall outside of this area and, therefore, the yellow keypoint is not matched.

On the bottom images, we see the opposite scenario. In this case, we find the same scene as before, and we want to match the yellow keypoint in (c) to the corresponding keypoint in (d). Therefore, we set a bigger window in (d), centred at the position of the yellow keypoint in (c), and we search for all the keypoints within the area of the window. As we see in the images, there are four different keypoints inside the window, but only two of those share the same information as the yellow one in (c). Therefore, the other two are discarded, but we face the problem of having two possible matches for the yellow keypoint. In this case, the algorithm would correctly choose the green one, because its distance to the position of the yellow one is smaller. However, a big window like the one seen here could also provide issues if, for instance, from (c) to (d) the bottom object was to disappear. In this hypothetical case, the yellow keypoint would be incorrectly matched to the red one.

With this, we can conclude that the adequate window size for the feature matching function depends on the specific characteristics of each game in particular (such as the speed of the objects’ movements or whether they get bigger or smaller throughout the sequence). However, we can establish that between two consecutive frames with a step value of 1, the objects’ position offset will be very small, so a smaller window size will be enough for a correct feature matching, while for bigger step values and, thus, bigger position offsets, larger windows are required. Therefore, for this project we have opted for a value of 5 for the parameter \( \text{win} \) which, in turn, results in a \( 10 \times 10 \) window around the keypoint location.

5.3 Overall performance of the program

In the first two sections of this chapter we have reviewed the two variable parameters that have a bigger impact on the functionality of the program: the threshold value for the SIFT algorithm and the window size for the feature matching function. In this section, we present a more general overview on the global performance of the program, with results for each step of the process on different games, in order to have a general view on its functionality.

5.3.1 Finding the keypoints

As explained in section 5.1, a very important part of this project is to correctly find the keypoints of each image in the sequence. The number of keypoints found strongly depends on the value of the threshold, so the first step was to find the most accurate default value for the threshold. Once done, we now present an overview on execution times according to the overall number of frames we take from each game.
### Table 5.2: Performance of the SIFT algorithm

<table>
<thead>
<tr>
<th>Game</th>
<th>step</th>
<th>Number of frames</th>
<th>Average keypoints</th>
<th>Time per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Air Raid</strong></td>
<td>1</td>
<td>858</td>
<td>40.21</td>
<td>0.288</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>172</td>
<td>40.41</td>
<td>0.266</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>18</td>
<td>41.22</td>
<td>0.282</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>9</td>
<td>43.56</td>
<td>0.276</td>
</tr>
<tr>
<td><strong>Centipede</strong></td>
<td>1</td>
<td>1821</td>
<td>138.05</td>
<td>0.334</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>365</td>
<td>138.05</td>
<td>0.285</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>37</td>
<td>138.12</td>
<td>0.263</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>19</td>
<td>136.89</td>
<td>0.268</td>
</tr>
<tr>
<td><strong>Demon Attack</strong></td>
<td>1</td>
<td>1566</td>
<td>22.54</td>
<td>0.263</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>314</td>
<td>23.02</td>
<td>0.290</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>32</td>
<td>25.59</td>
<td>0.258</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>16</td>
<td>25.56</td>
<td>0.265</td>
</tr>
<tr>
<td><strong>Freeway</strong></td>
<td>1</td>
<td>955</td>
<td>31.81</td>
<td>0.298</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>191</td>
<td>32.38</td>
<td>0.280</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>20</td>
<td>31.40</td>
<td>0.293</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>10</td>
<td>30.60</td>
<td>0.288</td>
</tr>
<tr>
<td><strong>Star Gunner</strong></td>
<td>1</td>
<td>1587</td>
<td>45.95</td>
<td>0.268</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>318</td>
<td>46.11</td>
<td>0.263</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>32</td>
<td>46.81</td>
<td>0.276</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>16</td>
<td>47.00</td>
<td>0.279</td>
</tr>
</tbody>
</table>

In the table we observe how the SIFT algorithm takes an average of 0.278 seconds to find all the keypoints in every frame for a threshold value of 0.025. Even though these five games show quite a varying average number of keypoints between them, the average time that the SIFT algorithm takes per frame does not vary more than five hundredths of a second. Therefore, we can conclude that the SIFT algorithm performs quite well in terms of time elapsed in finding all the keypoints. However, an inevitable issue appears when we decrease the value of the step parameter, as the more frames we have in the analysis sequence, the more aggregate time it takes to process the whole sequence.

### 5.3.2 Matching the features

In section 5.2, we have presented our revised version of the feature matching function, called `findMatches`, and we have seen that it provides a significant improvement in the process. In this section, we test it and compare it to the original `siftmatch` function.
provided by [22]. In order to carry out this comparison, we present a table with the time it takes to both functions to perform the same matching, for ten different games, with a step of 50 between consecutive images in the frame sequence, and a window size of 5 (exclusive for our function).

<table>
<thead>
<tr>
<th>Game</th>
<th>siftmatches</th>
<th>findMatches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Time per</td>
</tr>
<tr>
<td></td>
<td>matches</td>
<td>matching</td>
</tr>
<tr>
<td>Alien</td>
<td>21.18</td>
<td>0.002</td>
</tr>
<tr>
<td>Boxing</td>
<td>31.94</td>
<td>0.009</td>
</tr>
<tr>
<td>Crazy Climber</td>
<td>186.55</td>
<td>0.008</td>
</tr>
<tr>
<td>Fishing Derby</td>
<td>95.63</td>
<td>0.003</td>
</tr>
<tr>
<td>Gopher</td>
<td>46.50</td>
<td>0.003</td>
</tr>
<tr>
<td>Pitfall!</td>
<td>115.71</td>
<td>0.003</td>
</tr>
<tr>
<td>Q*bert</td>
<td>356.09</td>
<td>0.021</td>
</tr>
<tr>
<td>Venture</td>
<td>172.33</td>
<td>0.008</td>
</tr>
<tr>
<td>Video Pinball</td>
<td>258.31</td>
<td>0.011</td>
</tr>
<tr>
<td>Wizard of Wor</td>
<td>207.68</td>
<td>0.009</td>
</tr>
</tbody>
</table>

Table 5.3: Feature matching comparison

In the table above we see that both functions find exactly the same amount of matches. However, as explained before, the original function does not apply any restriction to the process, so the keypoints of different objects that look the same is incorrect. In terms of time, our function takes, on average, three hundredths of a second more to match each pair of images.

With the information shown in this table, we can conclude that the significant improvement in the accuracy of the matches is far more important to the overall performance of the program implemented in this project and, thus, we believe our function comes closer to fulfil our goals.

5.3.3 Getting information about the objects

After finding the keypoints and matching their respective descriptors, we proceed to segmenting the images of the sequence into connected components, in order to find the objects on the scene. Next, we show examples of the result of applying the connected components algorithm described in section 4.3 previously, for the games Carnival (a), Double Dunk (b), Elevator Action (c), Q*bert (d) and Space Invaders (e).
As we can see, the function performs quite well, but there are some inconsistencies as well. In the cases with a clearly distinguished background and a series of independent objects, as in (a), (d) and (e), the algorithm finds all the connected components without presenting any issues. However, in the cases with some objects intersecting with others, as in (b) and (c), the function englobes several objects into one big object. For example, in (b), we can see two players, the area and the basket all counted as one object. This is due to the fact that the program only takes out the pixels that correspond to the background, but all the others are counted as the same connected component as long as they are adjacent to each other. Therefore, we will need to take into account the fact that the games with objects that intersect between them are going to cause some issues in the correct functioning of the code, because some of the objects will not be detected. From those objects that are detected, the next step in the program is to get crucial information about them in a pixel level. This is done with the `regionprops` function, as explained in the previous chapter. Once the connected components are determined, this function returns a structure for each object consisting on several fields with information about that object in particular. For instance, and following with the examples shown in Figure 5.6, we can get all the information from the tenth object in the 101st frame of the game Carnival (a) as a structure shaped like this:

```
Area: 140
Centroid: [118.5000 70.2857]
BoundingBox: [110.5000 63.5000 16 14]
PixelIdxList: [140x1 double]
PixelList: [140x2 double]
isStatic: 'No'
```

From this information, and in this particular case, we can extract that this is a mobile object, centred at position [118.5000 70.2857] and with a total area of 140 pixels. The specific pixels that shape this object are listed within the field `PixelList`, as well as the linear indices of the matrix, which are listed within the field `PixelIdxList`. Lastly, we also obtain the coordinates of the upper left corner of the bounding box that encloses the object, as well as the width and height of this box. This information is later used by the program in order to classify the objects, track their movements and display the results.
5.3.4 Classifying the objects

Once the objects have been segmented from the scene and the necessary information has been extracted from each one of them, the program classifies them between mobile and static with the classifyObjects function. We need to take into account that, for this step of the process, we are restricted by the results of the previous one, in the sense that an object that is not well segmented from the screen (e.g. the case shown in Figure 5.6 (b) with the basketball players) will not be considered when extracting their information. Next, we show the results of applying the object classification algorithm in the games Asteroids (a), Berzerk (b), Double Dunk (c), Montezuma’s Revenge (d), and Space Invaders (e), with red bounding boxes for static objects, and green for mobile ones.
Figure 5.7: Examples of object classification

In these images we see an extent of what we have explained in the previous section: those objects that are not correctly detected by the connected components algorithm are also not well classified, as we can see in Figure 5.7 following with the example of Double Dunk (c). We observe a large, static object that covers the majority of the screen, consisting of the area, the basket and the three players that are intersecting with some point of the area. On the other hand, we can see a player and the ball correctly detected and classified as mobile objects. In the cases of Berzerk (b) and Montezuma’s Revenge (d), we can see a small problem that only affects visualisation: some large objects do not have a rectangular shape, but the bounding boxes that enclose them do, so visually we seem to have superposed objects. However, this is just a matter of display, and the actual result of the algorithm is correct. In the remaining cases of the games Asteroids (a) and Space Invaders (e), we can see a combination of a good performance by the algorithm and a clear display of the results, in the sense that all the objects are well classified, and we do not have any objects intersecting with the others, which could, in turn, cause some issues to the final representation of the results.

5.3.5 Tracking the objects’ movements

The final part of the program tracks the coordinates of the centroid of the selected objects throughout the frames, in order to have a graphic representation of how the objects in the
scene move. This representation is quite useful when studying the objects’ movements, as it allows us to see their evolution over time. In cases of a game with a considerable amount of objects, for instance, this function allows us to check whether these objects have a common way of moving around the scene, or instead they all go their own way. In Figure 5.8 below we present an example of the tracking of three different objects’ trajectories throughout the course of 100 frames.

![Figure 5.8: Example of the evolution of three objects’ trajectories](image)

In this particular example, we see an interesting case of three objects. The first one, in red, is static throughout the whole 100-frame sequence, while the other two are mobile. What is more interesting is the fact that these two mobile objects follow the same pattern, but in different speeds. In particular, the orange one goes slower compared to the blue one, but they both move at the same instants in time, allowing us to infer that these two objects’ movements are part of a single type of movements within the game.

We do have to take into account, though, that this algorithm strongly depends on the success of the object segmentation function, because if an object is not well identified due to intersections with other objects, the information of the centroid is also not well retrieved, and, thus, its position cannot be tracked properly. Therefore, we can resolve that the tracking algorithm works correctly when all the objects are well distinguished from the background of the images and do not intersect with other objects, while its performance can be affected otherwise.

With this, we conclude this chapter dedicated to the analysis of the program’s performance. In the next chapter we wrap up this project by reflecting on the work done throughout the process, as well as the achievements that have been accomplished and the future work that can be carried out after this project.
6 CONCLUSION

In this final chapter, we conclude this report by providing an overview on the whole project. We begin by reviewing the results and putting them in contrast with the objectives that we have described in section 1.3 of the introductory chapter, and later we present some of the possibilities for future work regarding the scope of this project.

6.1 Overview of the project

In this project, we have presented an alternative approach to the problem of optimizing the performance of the Arcade Learning Environment, which consists on extracting representative features from the screen of the game through several image processing techniques. The main goal is to provide enough information to learn the symbolic features of these specific kind of images, and use it to implement planning algorithms that are able to play Atari video games.

The project has been structured as a progressive process, starting from the basics and building up towards the reaching of the main goals. Hence, we have started by presenting the group of people that have worked in this project, as well as the main motivations and objectives of the project. In order to start building up, we have chosen to carry out a preliminary study about the world of video games, briefly covering its history and the industry behind it. This project is based on Atari 2600 video games, therefore we have introduced this console and its games, along with the ALE framework, two key pieces of this project. After this introduction, we have presented some of the most important and useful digital image processing techniques, and then we have adapted them in the shape of an analysis algorithm through the implementation of a MATLAB program. Lastly, we have evaluated the results of this algorithm. Having a library of 57 different Atari video games supported by the ALE has allowed us to test our program in very different circumstances. With this, we have been able to determine an accurate SIFT threshold value of 0.025, which adapts well to all the games, taking into account the total amount of keypoints that it finds, but also the relevance of those keypoints and the execution cost of the program. Another parameter that we have defined is the window size that allows for a well-performing feature matching algorithm. This window has a crucial effect on the result of the matches that the program finds, therefore the value of this parameter needs to be well adjusted to the games’ characteristics. We have defined a default value of 5 for this parameter—which results in a $10 \times 10$ pixel window around the keypoint—, assuming a small step between consecutive frames, which in turn results in small position offsets. However, for bigger steps between frames, we recommend to increase this value, as movements can become more noticeable and, hence, the matching algorithm’s performance could decrease. Finally, we have reviewed the overall performance of the rest of the program, including the connected components algorithm and the effects it has into finding the objects in the scenes.
Therefore, and after the analysis of the algorithm’s performance presented in the fifth chapter of this report, we can state that the proposed algorithm performs quite well when the conditions are favourable, with clearly distinguished objects with a small movement between consecutive frames of a sequence, and with the adequate values for the SIFT threshold and window size parameters. However, its performance is affected when these conditions are not met, and the accuracy of its results is reduced. In the next section, we evaluate the achievements that we have accomplished during the whole process, by looking back at the original objectives we set in the first chapter.

6.1.1 Achievements

The first thing to do when facing the development of a project is to set some goals and objectives to fulfil. At the end of this process, it is good practice to look back at those objectives and see which of those have been accomplished and which of those need more work. Hence, in this project we have:

- Completed a brief but concise study on video games, their history through the years and the industry that takes care of them, in order to set the foundation for the study of the Atari 2600 console and its video games and, consequently, introduce the concept of the Arcade Learning Environment, the framework around which this project is based.

-Introduced the concept of digital image processing and presented a study of a few of the most important techniques in this field of research from a mathematical point of view. With this study, we aim to provide a conceptual insight on the techniques that we have used in our analysis process, which can be then related to the implementation of these techniques.

- Adapted these digital image processing techniques into an analysis algorithm and implemented it as a MATLAB program. This program consists on a main script and several functions, with the aim of performing the necessary tasks in order to extract the desired features from a library of 57 different Atari video games, provided in the form of ROM files.

- Analysed the results of the proposed program, finding the most adequate values for the SIFT threshold and window size parameters, and reviewing the overall performance of the program in terms of accuracy, execution times, and visualisation of the graphic results.

- Set a groundwork that provides the necessary tools to carry out the following steps of the process from which this project is just an initial step. For this, we have put a special emphasis on describing the Arcade Learning Environment, and how to install it, compile it, and use it with the purpose of extracting frames from the Atari video games. We have also put a lot of effort in detailing the different image processing techniques that have been applied during the process, as well as our proposed algorithm and its MATLAB implementation. With this, we intend to provide accurate guidance to future
projects that seek to continue forward with the present work, or instead to give an alternate approach to the same problem.

Therefore, we can conclude this section by stating that all the initial objectives that we proposed for this project have been accomplished, and that we have provided a first step towards the final goal within the scope of this project, as described in sections 1.2 and 1.3 of the first chapter.

6.2 Future work

As mentioned previously in section 1.3, this project is just the foundation towards the goal of learning a symbolic structure of the Atari video games, which can be taken advantage of through planning algorithms. Therefore, there is a great amount of work that can be carried out from this point onwards, on two separate paths: to improve the work of this project, and to go further in order to reach the main purpose from which this project is just a first step. In this section, we detail some of the improvements that can be addressed in the near future.

In terms of expanding the analysis and improving the results obtained in this project, future work includes:

- To combine the feature matching and connected components algorithms in order to match groups of keypoints that belong to a specific object.
- To improve the connected components algorithm so that it takes into account objects composed by more than one colour.
- To optimize the classification function in order to reduce execution costs.
- To implement a pattern recognition algorithm in order to classify objects by their type or role within the video games.
- To extract new features from the screen of the games in order to provide more extensive information about the scene.

On the other hand, the scope of this project goes beyond digital image processing, so future work to be performed includes:

- To apply the information about the games extracted from this project into the learning process and to develop planning algorithms that play Atari video games.
- To carry out a common project that includes and combines both the image analysis and artificial intelligence learning process.
Bibliography


https://en.wikipedia.org/wiki/Video_game


https://it.wikipedia.org/wiki/Turochamp


https://en.wikipedia.org/wiki/Analog_stick


https://en.wikipedia.org/wiki/Video_game_industry


https://en.wikipedia.org/wiki/Center_of_mass

[38] Nüremberg, R. "Calculating the area and centroid of a polygon in 2D". 2013. Imperial College London. Web.

Web. 07 June 2016. 
https://en.wikipedia.org/wiki/Minimum_bounding_box

APPENDICES
APPENDIX 1: SUPPORT DATA

This appendix is intended to present the complete information that has been used in several graphics throughout this report. We divide it into four sections, each destined to provide all the data for a specific graphic, in order to support the information presented in certain points of the report.

Section A

Here we present a table with all the data corresponding to the graphic in Figure 2.2 and, consequently, Figure 2.3 on section 2.1.1 in the second chapter of the report. This data gives all the details on worldwide sales of the most successful consoles of the five biggest companies in the video game industry.

<table>
<thead>
<tr>
<th>Company</th>
<th>Console</th>
<th>Millions of units sold worldwide</th>
<th>Aggregate millions of consoles sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nintendo</td>
<td>NES / SNES</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Game Boy</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nintendo 64</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GameCube</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wii</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NDS / N3DS</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wii U</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Sony</td>
<td>PlayStation</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PlayStation 2</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSP</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PlayStation 3</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSVita</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PlayStation 4</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td>XBOX</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XBOX 360</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XBOX One</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Sega</td>
<td>Master System</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Genesis</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Game Gear</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pico</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saturn</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreamcast</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Atari</td>
<td>Atari 2600</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atari 5200</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atari Lynx</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table A1.1: Worldwide sales by company and console
Here we present a table with all the data corresponding to the graphic in Figure 2.4 on section 2.1.2 in the second chapter of the report. This table provides the information about the worldwide revenue of the video game industry between 1992 and 2015, with the specific values of each year, and also the 2016 inflation-adjusted values to put them into perspective.

<table>
<thead>
<tr>
<th>Year</th>
<th>Worldwide revenue (US Billion Dollars)</th>
<th>2016 Inflation-adjusted Worldwide revenue (US Billion Dollars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992</td>
<td>28.1</td>
<td>47.9</td>
</tr>
<tr>
<td>1993</td>
<td>29.6</td>
<td>49.0</td>
</tr>
<tr>
<td>1994</td>
<td>29.3</td>
<td>47.3</td>
</tr>
<tr>
<td>1995</td>
<td>29.3</td>
<td>46.0</td>
</tr>
<tr>
<td>1996</td>
<td>44.0</td>
<td>67.1</td>
</tr>
<tr>
<td>1997</td>
<td>25.9</td>
<td>38.5</td>
</tr>
<tr>
<td>1998</td>
<td>30.0</td>
<td>44.0</td>
</tr>
<tr>
<td>1999</td>
<td>39.4</td>
<td>56.6</td>
</tr>
<tr>
<td>2000</td>
<td>36.4</td>
<td>50.2</td>
</tr>
<tr>
<td>2001</td>
<td>35.0</td>
<td>47.3</td>
</tr>
<tr>
<td>2002</td>
<td>37.8</td>
<td>50.3</td>
</tr>
<tr>
<td>2003</td>
<td>33.2</td>
<td>43.2</td>
</tr>
<tr>
<td>2004</td>
<td>44.9</td>
<td>56.9</td>
</tr>
<tr>
<td>2005</td>
<td>37.6</td>
<td>46.1</td>
</tr>
<tr>
<td>2006</td>
<td>42.5</td>
<td>50.5</td>
</tr>
<tr>
<td>2007</td>
<td>61.3</td>
<td>70.8</td>
</tr>
<tr>
<td>2008</td>
<td>67.5</td>
<td>75.3</td>
</tr>
<tr>
<td>2009</td>
<td>60.4</td>
<td>67.2</td>
</tr>
<tr>
<td>2010</td>
<td>62.7</td>
<td>68.3</td>
</tr>
<tr>
<td>2011</td>
<td>65.0</td>
<td>69.0</td>
</tr>
<tr>
<td>2012</td>
<td>63.0</td>
<td>65.7</td>
</tr>
<tr>
<td>2013</td>
<td>76.0</td>
<td>79.2</td>
</tr>
<tr>
<td>2014</td>
<td>81.5</td>
<td>82.4</td>
</tr>
<tr>
<td>2015</td>
<td>91.5</td>
<td>92.4</td>
</tr>
</tbody>
</table>

Table A1.2: Annual worldwide revenue of the industry
Section C

Here we present a table with all the data corresponding to the graphic in Figure 2.7 on section 2.2 in the second chapter of the report. This table provides the information about the all-time best-selling Atari 2600 video games, with their year of worldwide release and the number of copies sold around the globe.

<table>
<thead>
<tr>
<th>Game</th>
<th>Year of worldwide release</th>
<th>Millions of copies sold worldwide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pac-Man</td>
<td>1980</td>
<td>7</td>
</tr>
<tr>
<td>Pitfall!</td>
<td>1982</td>
<td>4</td>
</tr>
<tr>
<td>Asteroids</td>
<td>1981</td>
<td>3.8</td>
</tr>
<tr>
<td>Missile Command</td>
<td>1980</td>
<td>2.5</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>1978</td>
<td>2</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>1982</td>
<td>2</td>
</tr>
<tr>
<td>E.T.</td>
<td>1982</td>
<td>1.5</td>
</tr>
<tr>
<td>Adventure</td>
<td>1979</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A1.3: Best-selling Atari 2600 games

Section D

The following table shows each of the 62 games that are supported by the current release of the ALE (v.0.5.1). Those games marked with a * are supported by the ALE, but are not included in the library of ROMs and, hence, have not been tested.

<table>
<thead>
<tr>
<th>Name of the game</th>
<th>Code-word supported by the ALE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air Raid</td>
<td>air_raid</td>
</tr>
<tr>
<td>Alien</td>
<td>alien</td>
</tr>
<tr>
<td>Amidar</td>
<td>amidar</td>
</tr>
<tr>
<td>Assault</td>
<td>assault</td>
</tr>
<tr>
<td>Asterix</td>
<td>asterix</td>
</tr>
<tr>
<td>Asteroids</td>
<td>asteroids</td>
</tr>
<tr>
<td>Atlantis</td>
<td>atlantis</td>
</tr>
<tr>
<td>Bank Heist</td>
<td>bank_heist</td>
</tr>
<tr>
<td>Battle Zone</td>
<td>battle_zone</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>beam_rider</td>
</tr>
<tr>
<td>Berzerk</td>
<td>berzerk</td>
</tr>
</tbody>
</table>
Bowling  bowling
Boxing  boxing
Breakout  breakout
Carnival  carnival
Centipede  centipede
Chopper Command  chopper_command
Crazy Climber  crazy_climber
Defender  defender
demon_attack  demon attack
double_dunk  double dunk
elevator_action  elevator action
Enduro  enduro
Fishing Derby  fishing_derby
Freeway  freeway
Frostbite  frostbite
Gopher  gopher
Gravitar  gravitar
Hero*  hero
Ice Hockey  ice_hockey
James Bond*  james bond
Journey Escape  journey escape
Kangaroo  kangaroo
Krull  krull
Kung Fu Master  kung_fu_master
Montezuma’s Revenge  montezuma_revenge
MS. Pacman  ms_pacman
Name This Game  name_this_game
Phoenix  phoenix
Pitfall  pitfalls
Pong*  pong
Pooyan  pooyan
Private Eye  private_eye
Q*bert  qbert
Riverraid  riverraid
Road Runner  road_runner
Robo Tank*  robo_tank
Seaquest  seaquest
Skiing  skiing
Solaris  solaris
Space Invaders  space_invaders
Star Gunner  star_gunner
<table>
<thead>
<tr>
<th>Game</th>
<th>Code-word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tennis</td>
<td>tennis</td>
</tr>
<tr>
<td>Tetris*</td>
<td>tetris</td>
</tr>
<tr>
<td>Time Pilot</td>
<td>time_pilot</td>
</tr>
<tr>
<td>Tutankhamun</td>
<td>tutankham</td>
</tr>
<tr>
<td>Up N’ Down</td>
<td>up_n_down</td>
</tr>
<tr>
<td>Venture</td>
<td>venture</td>
</tr>
<tr>
<td>Video Pinball</td>
<td>video_pinball</td>
</tr>
<tr>
<td>Wizard of Wor</td>
<td>wizard_of_wor</td>
</tr>
<tr>
<td>Yars Revenge</td>
<td>yars_revenge</td>
</tr>
<tr>
<td>Zaxxon</td>
<td>zaxxon</td>
</tr>
</tbody>
</table>

Table A1.4: Games and code-words supported by the ALE
APPENDIX 2: SURVEY AND RESULTS

In the second chapter of this report, we provide an introduction to video game history and the industry behind their production, development and promotion, as well as a more detailed look at one of the most successful home consoles of all time: the Atari 2600. In order to put this information into perspective, we have asked a diverse group of people to answer a short survey in order to get a better understanding on how their experience with video games is, as well as their knowledge of the Atari 2600. In this appendix, we provide a look in depth to the questions of the survey, as well as a graphic representation of the workflow according to the answers given to some specific questions.

Section A

Section 2.3.1 of the second chapter of the report gives a detailed look at the questions asked to the respondents. Here we provide the exact questions, separated into different sections depending on certain answers to some specific questions.

First of all, tell me some things about yourself

1. Gender
   - Male
   - Female

2. Age
   - < 20
   - 20 – 25
   - 25 – 30
   - 30 – 35
   - 35 – 40
   - > 40

3. What is your occupation?
   - I am studying
   - I have a job
   - I am studying and I have a job
   - Nothing at the moment

4. Do you play video games?
   - Yes
   - No
Let's see how much of a gamer you are...

*NOTE: this section will only appear to those who answer *Yes* to the previous question *Do you play video games?*

5. How many hours a week do you spend playing videogames?
   - 0 – 5
   - 5 – 10
   - 10 – 15
   - > 15

6. What platform/s do you use?
   *NOTE: multiple answers are accepted in this question.*
   - PlayStation
   - XBOX
   - PC
   - Nintendo
   - Wii
   - Smartphone
   - Tablet

7. How do you rate the importance of videogames in our daily lives?
   *NOTE: on a scale from 0 (not important) to 10 (very important).*
   - 0 1 2 3 4 5 6 7 8 9 10

8. Do you know the Atari 2600 console?
   - Yes
   - Maybe, it rings a bell
   - No, never heard of it

Why not?

*NOTE: this section will only appear to those who answer *No* to the previous question *Do you play video games?*

9. Why don't you play videogames?
   - I don't have time for them, but I'd like to play
   - I tried them, but they are boring
   - I'm not interested
I don't like them

10. How do you rate the importance of videogames in our daily lives?

*NOTE:* on a scale from 0 (not important) to 10 (very important).

0 1 2 3 4 5 6 7 8 9 10

11. Do you know the Atari 2600 console?

Yes

Maybe, it rings a bell

No, never heard of it

**About the Atari 2600 console**

*NOTE:* this section will only appear to those who answer *Yes* to the previous question *Do you know the Atari 2600 console?*, whether or not they play video games.

12. What is your knowledge about this console?

*NOTE:* multiple answers are accepted in this question.

- I own/owned one
- Someone in my family owns/owned one
- Some friend of mine owns/owned one
- I have only heard of it, but never seen one in real life
- Other

13. Which of the following Atari 2600 games do you know?

*NOTE:* multiple answers are accepted in this question.

- Alien
- Freeway
- Montezuma’s Revenge
- Ms. Pac-Man
- Road Runner
- Space Invaders
- Tennis
- Video Pinball
- Zaxxon
- None of them

14. Have you ever played games on this console?

- Yes, and I still do
- Yes, but not anymore
No, never

**How is/was your experience with it?**

*NOTE: this section will only appear to those who answer *Yes* to the previous question *Have you ever played games on this console?*

15. How would you rate your experience with this console?

*NOTE: on a scale from 0 (very bad) to 10 (very good).*

0  1  2  3  4  5  6  7  8  9  10

16. Which of the following Atari 2600 games have you played?

*NOTE: multiple answers are accepted in this question.*

- Alien
- Freeway
- Montezuma’s Revenge
- Ms. Pac-Man
- Road Runner
- Space Invaders
- Tennis
- Video Pinball
- Zaxxon
- None of them

**About my project**

*NOTE: this section will appear to every respondent, at the very end of the survey, independently of their previous answers.*

17. What do you think about the idea of a computer playing video games on its own?

- Very cool
- Seems interesting
- I don’t see the point
- I don’t care

18. How would you rate your knowledge on image processing?

*NOTE: on a scale from 0 (not a clue) to 10 (I’m an expert).*

0  1  2  3  4  5  6  7  8  9  10
19. Rate the following image processing techniques according to how much you believe they could help improve the computer's performance when playing a video game

**NOTE:** on a scale from 0 (not important) to 5 (very important). Also, it is not compulsory to provide an answer for every section of this question, if the user does not know what to say, he can leave it blank.

- Background segmentation
- Object detection
- Object classification
- Counting objects
- Tracking of movements
Section B

As explained in section 2.3.1, some sections of the survey depend on what the respondent answers to some specific questions. In this section we present a flowchart that depicts the possible outcomes of the survey according to the answers to these specific questions. In blue we find those questions that have a unique answer, while in green we show those questions whose answer will affect the proceeding of the survey.

Figure A2.1: Flowchart of the survey’s questions
APPENDIX 3: ALE’S OPTIONAL PARAMETERS

This appendix provides an insight into the parameters that can be specified when calling the ALE from the terminal, in order for it to play a game. Next, we present the result of running the −help command through this line:

User/ALE> ./ale −help

Which results in the following explanation of each possible input parameter of a call to the ALE from the terminal:

Main arguments:
- −help -- prints out help information
- −game_controller [fifo|fifo_named] (default: unset)
  Defines how Stella communicates with the player agent:
  - 'fifo': Control occurs through FIFO pipes
  - 'fifo_named': Control occurs through named FIFO pipes
- −random_seed [n|time] (default: time)
  Sets the seed used for random number generation
- −display_screen [true|false] (default: false)
  Displays the game screen
- −sound [true|false] (default: false)
  Enable game sounds

Environment arguments:
- −max_num_frames m (default: 0)
  The program will quit after this number of frames. 0 means never.
- −max_num_frames_per_episode m (default: 0)
  Ends each episode after this number of frames. 0 means never.
- −color_averaging [true|false] (default: false)
  Phosphor blends screens to reduce flicker
- −record_screen_dir [save_directory]
  Saves game screen images to save_directory
- −repeat_action_probability (default: 0.25)
  Stochasticity in the environment. It is the probability the previous
  action will repeated without executing the new one.

FIFO Controller arguments:
- −run_length_encoding [true|false] (default: true)
  Encodes data using run-length encoding

Misc. arguments:
- −ld [A/B] (default: B)
  Left player difficulty. B means easy.
- −rd [A/B] (default: B)
  Right player difficulty. B means easy.
APPENDIX 4: PSEUDOCODE

Throughout the development phase of this project, we have implemented several MATLAB functions to help apply the necessary algorithms in order to carry out the games’ analysis. In this chapter we provide the pseudocode for three of the most important of these functions.

Section A

Once all the SIFT keypoints have been located, the next step is to match their respective sets of descriptors between consecutive frames of the game’s sequence. Here we provide the pseudocode for the findMatches function, our revised version of the siftmatch function provided by [22].

Algorithm 1: Find matching features in a set of descriptors

**Input:**
- `keypts`, set of keypoints
- `desc`, respective descriptors of each keypoint
- `win`, window size

**Output:**
- `matches`, pairs of matches between consecutive frames

```plaintext
for each set of keypoints do
    get the desc of the keypts that fall within the window
    // Compare the current keypoint with the ones within the window
    for each pair of current and studied descriptors do
        d = \sqrt{\text{sum}((\text{current}_i - \text{studied}_i)^2)}
        // Get the minimum distance
        n = \text{min}(d)
        assign the corresponding pair of descriptors to the output matches
```

Section B

In order to identify all the objects in a scene, we have implemented the connected components algorithm, which segments the image into regions of adjacent pixels, which in turn allows us to isolate the objects from the background. Here we provide the pseudocode for the `findConnComp` function, presented in section 4.3 of the report.

Algorithm 2: Segment an image into connected components

<table>
<thead>
<tr>
<th>Input:</th>
<th>A: original image matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>B: labelled matrix</td>
</tr>
<tr>
<td>Parameters:</td>
<td>v: background pixel value</td>
</tr>
</tbody>
</table>

select the background colour and assign it to v
for each location in A do
    if this location is not visited then
        mark that position as visited
        assign the unique ID count to B

        // Look at its 8 neighbours
        remove those pixels out of bounds
        remove those pixels already visited
        remove those pixels with a value equal to v

        add the remaining pixels to the current connected component

        increment count on 1 unit
Section C

After segmenting the screen into several objects and obtaining information about them, the algorithm classifies them according to their mobility. Here we provide the pseudocode for the classifyObjects function, presented in section 4.5 of the report.

Algorithm 3: Classify objects between static and mobile

Input: info: structure with objects’ information
Output: info: updated structure with objects’ information
Parameters: ratio: ratio between intersections and total area

for each frame do
    for each object in the frame do
        get the current object’s information
        for each object in the other frames do
            find intersections between the current object and the others
            intsc = greatest number of pixel intersections
            area = corresponding object’s area
            ratio = intsc/area
            if ratio ≥ 0.9 then
                assign the object as static
            else
                assign the object as mobile

APPENDIX 5: SUPPORT FIGURES

In this appendix we provide complementary examples to the one shown in Figure 5.2 in section 5.1.1 of the fifth chapter for five additional games.
Figure A5.1: Case of *Atlantis*
Figure A5.2: Case of Beam Rider
Figure A5.3: Case of Gravitar
Figure A5.4: Case of Phoenix
Figure A5.5: Case of Zaxxon