Motion Control for an Artificial Character teaming with a Human Player in a Casual Game

Carlos de Sousa Fraga

Dissertação para a Obtenção do Grau de Mestre em Engenharia Informática e de Computadores

Júri
Presidente: Prof José Manuel da Costa Alves Marques
Orientador: Prof Rui Prada
Vogal: Prof Rodrigo Ventura

Outubro de 2011
Acknowledgements

This work would not have happened if it was not for all the support given to me during its execution.

I would like to start by thanking my advisor, Rui Prada, and co-advisor, Francisco Melo, for all the patience, availability and outstanding expertise in subjects of utmost importance to this work. Based on their guidance and feedback I was able to overcome many challenges and make decisions that could otherwise be overlooked. I would also like to thank all my friends and co-workers who would voluntarily listen to whatever problem was troubling at that time and propose alternatives, even when they were not comfortable with the issue at hand. More often than not those alternatives were a trigger to a different train of thought that would culminate in a solution to said issue.

At last but certainly not least, I would like to express my utmost gratitude to my family and girlfriend for coping up with the constant incapacity to communicate with me for days, the last minute cancelations and occasional grumpiness. It was their support and understanding that kept me going through the whole process. I would also like to thank them for the effort put in trying to give me feedback on the written document, even though they have no academic knowledge on this subject.
Abstract

Agent interaction in a cooperative, physics based multiplayer videogame is an interesting, albeit relatively unexplored area of research. However, before tackling teamwork issues, an agent must be able to overcome several motion control and coordination challenges. Our work created an agent capable of interpreting the game world where it was inserted, planning actions taking into account its own capabilities, its teammate capabilities and the world state. It also acts according to these plans and is able to identify team coordination points in the game world.

In order to achieve this objective, 4 key concepts were identified. These concepts are, in order of relevance to this work, motion control, coordination, cooperation and teamwork. We analyzed several studies in diverse areas of research about these concepts and hereby present the result of that research.

The solution proposed in this document is based on existing tiered robotic control architectures and algorithms. We proceeded to integrate this solution in a NPC in the game Geometry Friends (an existent casual game) in order to evaluate its capacity to solve several types of challenges.

Keywords: motion control, coordination, cooperation, teamwork, agent, human player, casual game
Resumo

A interação entre agentes e jogadores humanos num jogo cooperativo baseado em coordenação motora é um tema de pesquisa interessante e pouco explorado. No entanto, antes de se poder explorar a vertente de equipa, o agente tem de ser capaz de ultrapassar individualmente uma série de desafios a nível de coordenação e controlo motor. Esta tese criou um agente capaz de interpretar o ambiente em que se encontra e planear as suas acções de acordo com as suas limitações, as do seu colega de equipa e o estado actual do mundo que o rodeia. É também capaz de identificar pontos em que é necessário trabalho de equipa.

Para atingir este objectivo identificamos quatro conceitos chave. Estes conceitos são, por ordem de relevância para este trabalho, controlo motor, coordenação, cooperação e trabalho de equipa. De modo a compreender melhor estes conceitos e a relação entre eles, estudamos trabalhos de diversas áreas de pesquisa e apresentamos os resultados dessa pesquisa neste documento.

A solução por nós apresentada baseia-se em algoritmos e arquitecturas existentes na robótica. No entanto, várias adaptações tiveram de ser aplicadas de modo a que a nossa solução por camadas conseguisse suportar os conceitos previamente indicados. Esta solução foi posteriormente aplicada ao jogo Geometry Friends de modo a podermos testar o agente e a sua eficácia em relação ao controlo motor, coordenação e cooperação.

Palavras chave: Controlo motor, coordenação, cooperação, trabalho em equipa, agente, jogador humano, jogo casual
Contents

Abstract \hspace{1cm} v

Resumo \hspace{1cm} vii

List of Figures \hspace{1cm} xii

List of Tables \hspace{1cm} xv

Terminology and Acronyms \hspace{1cm} xvi

1 Introduction \hspace{1cm} 1
  1.1 Context and Motivation \hspace{1cm} 1
  1.2 Dissertation Statement \hspace{1cm} 2
  1.3 Document Outline \hspace{1cm} 2

2 Background \hspace{1cm} 5
  2.1 Games Using Cooperative NPC’s \hspace{1cm} 5
  \hspace{1cm} 2.1.1 Turn Based Strategy (TBS) \hspace{1cm} 5
  \hspace{1cm} 2.1.2 Real Time Strategy (RTS) \hspace{1cm} 5
  \hspace{1cm} 2.1.3 Role-Playing Games (RPG) \hspace{1cm} 6
  \hspace{1cm} 2.1.4 First Person Shooters (FPS) \hspace{1cm} 7
  \hspace{1cm} 2.1.5 Sports Games \hspace{1cm} 7
  \hspace{1cm} 2.1.6 Beat ’em ups \hspace{1cm} 8
  \hspace{1cm} 2.2 The Game: Geometry Friends \hspace{1cm} 9

3 Related Work \hspace{1cm} 13
  3.1 Teamwork \hspace{1cm} 14
  3.2 Cooperation \hspace{1cm} 15
# CONTENTS

3.3 Coordination ........................................... 17
3.4 Motion Control ........................................... 20

4 Proposed Solution ........................................ 27

4.1 Agent Overview ........................................... 27
4.2 Agent Architecture and Modules .......................... 29
  4.2.1 World Model .......................................... 30
  4.2.2 Navigational Graph Generation ....................... 31
    4.2.2.1 Regular Nodes Generation .................... 34
    4.2.2.2 Corner Nodes Generation ..................... 35
    4.2.2.3 Generation of Paths Between Nodes ............ 36
    4.2.2.4 Node Height Determination ................... 38
  4.2.3 Pathfinding ......................................... 40
  4.2.4 Teamwork ........................................... 40
  4.2.5 Cooperation ........................................ 41
    4.2.5.1 Objectives sequence determination ............ 41
  4.2.6 Task Determination .................................. 45
  4.2.7 Coordination ....................................... 46
  4.2.8 Motion Control layer ................................ 47
    4.2.8.1 Predicting an action result .................. 48
4.3 Testing Tools .......................................... 49
  4.3.1 Level Editor ........................................ 49
  4.3.2 Graph Viewer ....................................... 51
  4.3.3 Agent Data Viewer .................................. 51
4.4 Evaluation .............................................. 52
  4.4.1 Test Level 1 ....................................... 53
  4.4.2 Test Level 2 ....................................... 54
  4.4.3 Test Level 3 ....................................... 55
  4.4.4 Test Level 4 ....................................... 55
  4.4.5 Test Level 5 ....................................... 56
  4.4.6 Test Level 6 ....................................... 56
  4.4.7 Test Level 7 ....................................... 57
  4.4.8 Test Level 8 ....................................... 58
# List of Figures

2.1 Human and NPC controlled avatars and their movements . . . . . . . . . . . . 10
2.2 Example of a *Geometry Friends* level . . . . . . . . . . . . . . . . . . . . . . . . 11
3.1 Correlation between the different desired qualities in our agent (left) and *Ge-
ometry Friends* concepts (right) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 Example of a non-deterministic Tactic . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 $3T$ Architecture Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.4 Different hierarchies and relation between them. . . . . . . . . . . . . . . . . . 25
4.1 Example of a very simple level. $W_1$ and $W_2$ are nodes in the *Navigational
Graph*, while $C_1$ is the only Collectible, and thus objective, in the level. . . . . . 27
4.2 Sequence diagram representing the agent steps in each layer when executing
the level provided in figure 4.1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.3 High Level Architecture Diagram. Representation of different layers and com-
munication between them. Boxes next to bidirectional arrows represent data
entities shared by both layers. The dark gray box represents the whole agent. 30
4.4 Level where the Square controlled by agent could not create the path to the
remaining collectible. Path represented by red line. . . . . . . . . . . . . . . . . . 32
4.5 Level from figure 4.4 with first the version of *Navigational Graph*. White dots
represent graph nodes. Lines represent a unidirectional path from the node
they are touching to the node they are pointing to. Bidirectional paths are
represented by lines on differing nodes pointing to each other. Please bear in
mind that in closer nodes these lines overlap. Although they look like a single
line, they still represent a bi-directional path between two nodes. . . . . . . . . . . . 33
4.6 Again, level from figure 4.4, with the improved version of *Navigational Graph*.
In this case there is no noticeable advantage in using the improved version . . . 35
LIST OF FIGURES

4.7 Obstacle with nodes generated around each corner. .................................. 36
4.8 Example of a case where improved graph performs better than the original version. Notice how figure 4.8(a) has no link between the top and bottom, whereas figure 4.8(b) provides a path down .................................................. 37
4.9 Example of graph limitation. There is no path available to the collectible, even though it is possible for the ball to jump from one platform to the other .... 39
4.10 Example of a level with a specific order to catch the collectibles. For identification reasons, consider the first collectible the one with the top leftmost position. ................................................. 42
4.11 Diagram with a step by step representation of the objective sequence algorithm for level 4.10. .......................................................... 43
4.12 RAP entity details. ................................................................. 45
4.13 TaskState content. ................................................................. 46
4.14 State Machine for NavigateToC1 task. The value 1 represents a valid condition and the value 0 represents an invalid one. ................................. 47
4.15 Graphical representation of the MoveToW1 action timelines. Each bar represents a timeline. The values inside each segment are passed to the corresponding Actuator for their respective duration. ...................... 47
4.16 Screenshot from Level Editor with no tool selected. ..................... 50
4.17 Screenshot from Level Editor with a tool selected. .......................... 50
4.18 Level with Agent Data Viewer on. .................................................. 52
4.19 First Level. Single objective with cooperation ................................. 54
4.20 Level with specific objectives order .................................................. 54
4.21 Impossible level for square character ................................................. 55
4.22 Level with tight vertical passage ...................................................... 55
4.23 Level with specific order and tight horizontal passage ....................... 56
4.24 Level to test fast square morphing ................................................... 57
4.25 First level used to test circle and square ........................................... 58
4.26 Level to test simple circle jumps ...................................................... 59
4.27 Level that demonstrates error in node graph ........................................ 59
4.28 Another level with a graph limitation. Graph can be seen in figure 4.9 .... 60
List of Tables

4.1 Starting Tuple .......................................................... 42
4.2 Tuple representing level with collectible 2 caught. .................. 43
4.3 Tuple representing level with collectibles 2 and 3 caught with agent in a position where it is impossible to catch collectible 1. .................. 43
4.4 Equivalent tuples and their merge. .................................. 44
4.5 Objectives sequence .................................................... 44
Terminology and Acronyms

NPC Non Player Character - game character controlled by the computer. Also known as Artificial Character.
TBS Turn Based Strategy - genre of video games.
RTS Real Time Strategy - genre of video games.
RPG Roleplaying Game - genre of video games.
FPS First Person Shooter - genre of video games.
AI Artificial Intelligence.
Bot - term most commonly used in FPS's to refer to computer controlled characters.
SWAT Special Weapons and Tactics - Name given to a north american tactical unit responsible by performing high risk hostage rescues and counter terrorism operations.
FIFA Federation Interazione de Football Association - Organization responsible for regulating soccer and managing the World Cups.
Winger - term used in soccer to reference players whose field position lies on the sides of the field and are commonly responsible passing the ball for other players to score a goal.
RAP Reactive Action Package - Term used to refer a data entity used on our solution. Originally coined by R. James Firby[8] and later used by Bonasso[28].
1 Introduction

1.1 Context and Motivation

Video games have evolved a lot since their birth. As the hardware platforms where they performed evolved, so did the games themselves. They became more complex in all aspects, not only in performance consuming features (such as graphics and sound), but also in design (as in game mechanics, story). Video game development went from academic or experimental work done by a few people to multi-million dollar budget projects carried out by hundreds of people [1]. As they matured, so did the NPCs (Non-Player Characters) featured in them. There has been some sort of AI (Artificial Intelligence) in video games since the beginning. In fact, there was AI for games even before video games appeared\(^1\). It was only in the mid 70’s that games with single player mode started to emerge, featuring computer controlled enemies. The AI in these enemies was restricted to stored patterns, eventually evolving to movement patterns with random factors. Nowadays games are using NPCs as competitive adversaries to beat, enemies to destroy or challenges to overcome and as cooperative pets, allies or even teammates.

However, using NPCs as teammates with human players is an area in video game AI that is still fairly lacking. Besides being a very complex issue to solve, there are also heavy computational requirements on complex AI: since a game is usually composed of several CPU intensive features such as graphics [12] (and more recently physics), the processing power left for AI algorithms tended to be limited. That is one of the reasons developers often had to opt for light, script-based AI behavior which are severely lacking in terms of flexibility. Furthermore, in cooperative scenarios with human players, flexibility is a key aspect of NPC AI. Developers often compensate that lack of flexibility by giving the player direct control over the actions of

\(^1\)In the middle 50’s a checkers artificial intelligence program developed by A. Samuels had enough skill to challenge an amateur player [2].
the NPCs. While this works in a satisfactory way in some cases, it shifts unnecessary workload to the player, forcing him to command extra characters if he wants them to behave effectively. There are several scenarios where that may not be feasible at all: if a player has a whole group of allied NPCs to command, he will not be able to order each of them individually in an effective way in real-time.

1.2 Dissertation Statement

In order to create an agent that allows a NPC to interact with human players as a team in a casual game, a solution capable of achieving Teamwork, Cooperation, Coordination and Motion Control is required. Nevertheless, tackling all these concepts would broaden the scope of this work too much. Therefore we focused on making the agent capable of autonomous movement and reasoning within the game world, while never forgetting the long term goal of teamwork.

The result was a framework capable of full Motion Control and Coordination while providing some basic Cooperation features and support to future work in Teamwork. With this objective in mind an architecture was planned and an agent was implemented in a cooperative game called Geometry Friends in order to evaluate this solution. This game presented several challenges to our agent such as fine motor control, path finding, objectives sequencing and identification of team coordination points.

1.3 Document Outline

This document is structured throughout six chapters. Chapter 1, in which this section is included, introduces the theme of this thesis, its objective and how we proceed to explain what was achieved. Chapter 2 is an overview on game genres using cooperative NPCs, their most valued characteristics and limitations. Here we define the 4 most desired characteristics in our agent, Teamwork, Cooperation, Coordination and Motion Control. It is also here that Geometry Friends, the game where this solution was implemented, is described. In Chapter 3 we proceed to explore work done by the scientific community related to the 4 traits previously identified and how they relate to each other. Chapter 4 is where we describe the solution in detail. This chapter is divided by several sections such as the architecture, data entities used,
algorithms and testing tools. We also describe the evaluation process of our solution. We then follow with considerations about the contribution of our work as well as its limitations and future work in Chapter 5 and 6.
2 Background

2.1 Games Using Cooperative NPC’s

Many games have cooperative NPC’s but, as we can see in the following sections, the most desired characteristics in these vary greatly from genre to genre:

2.1.1 Turn Based Strategy (TBS)

These games are usually known for the personality of the AI-controlled opponents and the diplomacy options provided to interact with them. Although they are opponents, periods of cooperation between them and the player are common in order to gain advantage against common enemies. Examples of this type of games are the Civilization and Space Empires series. While the social interactions and relations between AI and human player are very complex, the actual cooperation is restricted to the sealing of treaties and exchange of goods. There is no actual teamwork or coordination in actions.

2.1.2 Real Time Strategy (RTS)

In RTS games such as Starcraft or Command & Conquer there are two types of AI:

- The limited AI in units controlled by the player, that enables them to travel from one place to another in the map (path-finding) and automatically attack nearby enemies;

- The AI in NPC-controlled ally factions.

The latter has the same power that human players have, controlling the units of their own faction. Although they are in the same team, there is no real teamwork; despite having a common objective (defeating the opponents), there is no coordination in the NPC actions, and therefore no cooperation, since the AI makes its decisions as if they were playing alone.
2.1.3 Role-Playing Games (RPG)

RPGs are games where the player controls a character in a fantasy world. Several NPCs reside in these fantasy worlds, with friendly or hostile behaviors. In many of these games the player has the capacity to have one or more allied NPCs that accompany him throughout the game (or part of it). Although they are different characters, very few RPGs actually have NPCs that do not need to be controlled by the player. Some games like *Diablo II* and *Fallout II* place emphasis on the motion control aspect of the NPC, providing it with autonomy. However, in both cases their coordination is very limited or inexistent. In the former, the player can hire a mercenary that acts autonomously. From a coordination perspective however, they are very limited. They have no consideration for the player actions, only relying on his position in order to orbit around him. In the latter game, a player can have a group of NPCs that engage in battles along with him. Although they behave autonomously, this behavior is determined by a set of tactical orders given by the player before combat. Also, there is no sense of coordination from the NPCs. They behave as they are ordered and it is up to the player to coordinate its own actions with them. Another approach is placing less emphasis in *motion control* while providing more *coordination*. Games such as *World of Warcraft* and *Guild Wars* feature NPCs that need to be explicitly ordered to do some actions, but also have some mechanisms that allow them react to player actions in certain conditions. For example, in *World of Warcraft* a certain type of player character can control a pet which, depending on a certain state previously set by the player, automatically attacks the same target the player is attacking. They also automatically stop attacking if their target is *crowd controlled* (term used to describe an enemy that has been put in an disabled stated for a long period of time, which is canceled by any damage done to it). *Guild Wars* took a hybrid approach, having two types of team NPCs: an autonomous type of NPC that is less effective, coordinating only at a basic conditional level (for example, if an ally is below 30% health, heal him) and a NPC that needs player input to use its abilities, but is much more effective at doing it (since it is the player coordinating both characters).

Playing those games makes a person quickly realize that having player controlled features in each NPC does not scale well, as one gets easily overburdened by all the decisions necessary to control the NPC’s efficiently. These games however both benefited greatly from the simple coordination mechanisms present in them.
2.1.4 First Person Shooters (FPS)

FPS games are played from a first person perspective and, as the name implies, usually consist on shooting enemies. They are usually about tactical positioning and fast reflexes. Some of these games feature squad mechanics making them impossible to play alone unless there is AI controlling the other squad members. These NPCs are usually known as bots. *Counter Strike* is one of those games. Thanks to its open framework where any user can develop their own game modifications, several different bots have been created. In most cases these are completely autonomous, choosing their own setup of weapons, armor and deciding where to go. The player can however, at any given time during the match, issue some orders such as “cover me”, “attack”, etc. These bots are usually very coordinated among themselves, but they do not take the human player into account unless an explicit order is given. In *Swat 4* the process is a bit different. Since the game is about simulating SWAT teams intervention in high risk situations with civilians and hostages, the tactical part of the game is much more important. Therefore NPC squad mates always wait for an explicit order from the player before taking important actions, like breaching into a room with terrorists.

As opposed to the former genre, in most FPS games reaction time is critical. Therefore, it is unacceptable to have NPCs dependent on player input. However, while autonomous NPCs can be effective, they only coordinate among themselves. To make them capable of coordinating with the player, there is the need for explicit communication.

2.1.5 Sports Games

Sports games with teams have similar requirements as the FPS games, albeit being less demanding on fast reaction times. For instance in soccer games like the *FIFA* series, the player only plays with one player at a time. The way the team acts is dictated by tactical choices before the match begins, player position and ball possession (or lack thereof) and limited player orders given by a single key press during the match (like "winger players run forward"). In these games we can see much more coordination without explicit communication. However, there is a particularity in this type of games: the human player actually controls every player in the field, one at a time (except the goalkeeper). This means that important actions like passing, shooting or tackling opponents are always done by the human player. Although this has to do with game design, it also minimizes the importance of the AI which is then restricted to simple player positioning in the field. In these games the AI focuses on coordination,
while cooperation and teamwork itself is all created by the player itself, since it is him who controls every interaction between team members. Motion control and coordination is mostly controlled by AI by the means of positioning related to the human controlled player.

2.1.6 Beat 'em ups

Beat 'em ups are games where the player controls a single character, having to fight through groups of increasingly stronger enemies. Some of these games have the option to play it cooperatively with another friend. However, Spider-Man: Friend or Foe takes this premise one step further, creating a game where the player controls a team of two: Spider man and a friend. When the game starts the player chooses one of many characters as a teammate. When getting into fights these become crucial not only because they do their share of combat (though they’re not as strong as the player, for balancing reasons), but also try to help the player by holding enemies while the player finishes them off or helping out while the player himself is holding or throwing enemies. Each teammate has different moves, and therefore different play styles. If the player wishes to have more control, he can also switch controls from Spider-Man to the teammate, letting Spider-Man become controlled by the game’s AI. There are other occasions where these team mates are necessary: sometimes the player needs to press two switches (by standing on top of them). When the player moves Spider-Man to the top of one of the switches, the computer controlled teammate will automatically search for the second one.

Of all the examples presented in the games section, this is the most similar in terms of coordination issues. It is a great example of how a NPC can be autonomous and behave consistently without the need for explicit communication. If that is still not enough, there is the option to jump in and control the second character. Even though there is no explicit communication between player and NPC, their coordination works and really feels like a team effort to the player. However, while the NPC is one of the main game features, the player does not have to rely on it to advance through most of the game (the exception are the occasional group switches).

The relative importance of each of these concepts depends on the type of game, since there is a different relation between human and NPC. When the NPC is a subordinate, he is expected to obey to the human player. That is not so when they are viewed as equals by the game (like when both are equally important members of a team), which brings us to the cooperation
aspect: it is very different from human to human or NPC to NPC interaction; human players are not completely predictable and can always act differently from what they convey to the NPC through explicit communication, meaning there is a need for flexibility and recovery in the NPC decision mechanisms. Lastly, there is the motion control issue itself.

Since NPC and human are working as teammates, they will have to cooperate in order to achieve their goals. Whereas two NPCs could exchange game data between them, with a human player the NPC has to be able to implicitly deduce most of that information if he wishes to achieve an effective level of coordination. For example, the NPC should be able to anticipate the player movement to a certain extent, like direction and/or speed, just by “observing” his movement pattern. A concrete example of these concepts is given in section 2.2 showing how they are mapped in Geometry Friends, the casual game where our agent will be implemented and tested.

Although this work is mostly focused on motion control and coordination issues, we expect to have an agent capable of controlling a NPC within the game virtual environment it is implemented in. We also expect it to be able to achieve the goals he is capable to accomplish individually, as well as coordinate with a human player in order to achieve simple tasks without the need to explicitly communicate with one another. The agent will also be able to complete simple levels through cooperation with the human player. Although the teamwork aspect of the agent will not be explored in this work there is special attention in the implementation of the solution, in order to facilitate future work in that area.

2.2 The Game: Geometry Friends

Geometry Friends is a casual game for two players. The game, running in a two dimensional world with a physics simulator, focuses on cooperation\(^1\). Each player controls one of two different geometric shapes (see Fig. 2.1):

- a circle, whose actions comprise moving sideways, jumping and increasing/decreasing its size (and consequently, its mass);

- a rectangle, whose actions comprise moving sideways, morphing its shape by stretching and contracting (while always maintaining the same area).

\(^1\)It is easier to understand how the game works by watching a video in http://geometryfriends.tumblr.com
Their objective is to complete each level of the game by catching every purple collectible scattered across the screen. However, in order to enforce cooperation, some of those diamonds are in places only reachable if both the circle and rectangle coordinate their actions (an example of a level can be seen in Fig. 2.2). In order to create those hard to reach places, there are a number of different platforms:

- regular platforms: black platforms against which both shapes collide.
- circle-only platforms: yellow platforms against which only the circle shape collides.
- rectangle-only platforms: green against which only the rectangle shape collides.

There are also moving platforms of the aforementioned types. Finally, the circle and the rectangle shapes also collide against each other. This means that, by combining their actions, they can achieve different results (for example, the rectangle can stretch while having the circle on top, allowing the latter to reach otherwise inaccessible spots). These joint actions are what make the game very intensive on coordination and cooperation because most levels are designed to be impossible to complete without at least one joint action.

When thinking about the game in terms of teamwork, cooperation, coordination and motion control, we can segment the agent influence in the game in the following way (see Fig. 3.1):

- teamwork - encases the process of beating the game and social interactions between NPC and human. All the knowledge kept through levels is related to this level
2.2. THE GAME: GEOMETRY FRIENDS

Figure 2.2: Example of a Geometry Friends level

- **cooperation** is the process of completing a single level. The whole tactical process of deciding which diamonds to catch first, as well as monitoring the game flow is part of cooperation.

- **coordination** is the execution of a single decision at cooperation level (i.e. catching a particular diamond) through synchronization of NPC and human actions belongs to this category.

- **motion control** - all the NPC shape controlling features belong to this category.

The biggest challenge of this game is the control of the shapes and coordinating the actions between both players. By watching two humans play the game, it was observed that they relied heavily on verbal communication not only at cooperation but also at coordination level. It was also possible to observe social interactions that had no relation to the game performance itself (like one player mocking the other because of incapacity to do a certain action).

Although the ultimate goal is having a NPC capable of behaving like another human player would, communicating and planning the level execution along with the other team member, the scope of this work is to have one of these shapes capable of moving autonomously, catching as many collectibles alone as possible and effectively identify those joint actions in order to complete the level.
3 Related Work

As it was previously mentioned, our desired agent must be capable of teamwork, cooperation, coordination and motion control. However, different definitions for this concepts have emerged during our research on each of them, sometimes even overlapping one another. Therefore, to keep this document consistent, we will present our work in terms of these four characteristics:

1. **teamwork** is the word used to describe the relationship between the NPC and the human player throughout the course of the game. This relationship is created through social awareness, social interaction, group dynamics, learning mechanisms and a common goal (both have the goal of beating the game).

2. **cooperation** occurs at the tactic context. It is the capacity to create and carry out a plan to achieve a set of objectives by both players. It requires decision-making mechanisms, constant monitoring and revision of objectives, as well as coordination.

3. **coordination** is the capacity to perform actions in consonance with the human player. This requires environment and human player awareness, synchronization and anticipation mechanisms.

4. **motion control** is the capacity of the agent to control its actions in the game world. This includes all mechanics needed to perform atomic actions, achieved through a simple game engine call, as well as tasks, comprised of several atomic actions achieved individually in a predetermined sequence (e.g, move right then jump).
CHAPTER 3. RELATED WORK

3.1 Teamwork

[tēmˈwɜːrk’]

noun,

Cooperative effort by the members of a group or team to achieve a common goal[35]

As previously stated in this document, teamwork between human and NPC is the ultimate goal of the solution we provide and, while it does not directly relate to motion control, it is crucial to understand how it works if we intend to create an agent that can achieve it. However, teamwork is a very broad concept. It has been studied for a long time in areas as different as Psychology, Economy and Artificial Intelligence. All these studies identify many factors that influence teamwork, such as the personality of each team member [11], individual goals [4], the surrounding environment [31], social roles [33, 32, 11] and group dynamics [19, 20, 21, 6, 14], just to name a few. Therefore, given the scope of this work, we chose to focus on the core concepts that enable an NPC to work effectively with a human player as a teammate, namely coordination and cooperation. There are several concepts that are connected to cooperation and they will also be discussed in this document, but first we need to explore the concept of team. This leaves us with a question:

Figure 3.1: Correlation between the different desired qualities in our agent (left) and Geometry Friends concepts (right)
3.2. COOPERATION

Why is teamwork different from simple cooperation?

As Cohen and Levesque pointed out, “We would not say there is any teamwork in ordinary automobile traffic, even though the drivers act simultaneously and are coordinated”[3]. As the above dictionary definition states it, teamwork differs from simple cooperation because there is a common goal every member of that team is working to achieve. Teamwork is different from cooperation in the sense that there are objectives common to the team members, objectives that are more important than the objectives of the individual itself. The aforementioned authors developed the concept of joint intention “which is formulated as a joint commitment to perform a collective action while in a certain shared mental state, as the glue that binds team members together”. In other words, in order to have a team, all members of that team must have at least some common goals (team objectives) and some sort of commitment to them (simply having a goal is not enough: one must commit to achieve it). One simple example of a joint intention is having two people carry a fridge to another room. They both have the intent to carry it, and they both intent to do it together, since it is too heavy and too large for a single person to achieve it. When their joint intention is materialized (i.e, when they actually carry the fridge), it becomes what one calls a joint action.

When considering the aforementioned authors definition of teamwork in [3], the difference between teamwork and cooperation becomes blurred. For this reason we differentiate by associating cooperation to decision-making and plan creation, and teamwork to social interaction by the team members. It can also be thought as a matter of scope. Teamwork requires a common ultimate goal (in Geometry Friends completing all levels) to exist, while cooperation only requires a situational common goal that allows both sides to achieve their own personal goals. So, while teamwork requires cooperation, the opposite is not necessarily true.

3.2 Cooperation

Another concept worth noting, as explored by Grosz and Kraus, is the concept of SharedPlan. A plan is defined by a set of beliefs and intentions. As similar as that might sound to joint
intentions, the authors defend that a team member does not have to know all intentions and actions of the other members. All he needs to know is whether their individual plans conflict at some point with his own plans, and all members must have SharedPlans as a base for their group actions [4]. Each team member has its own knowledge and beliefs on how something should be done and may have individual intentions about some of these sub actions. The authors also make distinction between Full Plans, where the agent already determined the whole course of actions to achieve it, and Partial Plans. In their work Grosz and Kraus not only formalized the concept of SharedPlan but also identified three major problems to deal with when using SharedPlans:

1. *The need to develop a way of resolving conflicts between plans.* If both agents need a determined resource to achieve their part of a goal, which one is going to use it?

2. *The need to develop better methods of agreement between members on a determined action.* Just because both know what action is does not mean they do it the same way. For example, lets say two agents have a SharedPlan: to cook a tuna pizza. They both know what the plan is, but maybe one does thin pizzas and the other does thick pizzas. The main issue here is which one to choose (because, obviously, the pizza can’t be both thick and thin).

3. *The need to understand how to make the agents communicate, in order to develop a better set of communication axioms.* Their work focused on developing a formal way of representing goals. Although they accounted for communication between agents about subtask completion and errors, there is still much to be improved in order to have a communication system that can serve humans and agents alike.

Even though coordination is not a key concern on the above paper, it is of utmost importance to understand how a team cooperates in order to be able to know how to coordinate its members. The problems that arise from their work confirm our definition of teamwork as more than just a formal representation of a team’s and its individual’s plans. As the word itself implies, while the common objectives and the commitment to achieve them is what keeps a team together, that representation does not provide a team with the mechanisms required to put those objectives to practice. Both the joint intentions and the SharedPlans theory focus on the creation of plans and objectives, which are cooperation issues, but neither provides solutions to the execution concerns (which are the scope of our work) such as how to carry out
those plans or how to keep the team members coordinated. Although they identified those issues, their resolution was beyond the scope of their work. Further studies based on these theories, such as STEAM [5], focus on solving these practical concerns, giving emphasis to coordination and communication.

3.3 Coordination

Coordination is the harmonious interaction of several individuals. However, coordination is not an individual characteristic, it is a group characteristic (or we could go even further and call it a state). Tambe et al had a similar concern when they developed a framework called STEAM[5]. This framework is based on the aforementioned joint intentions theory. However, as stated in his work, "while STEAM is founded on the joint intentions theory, practical operationalization has required it to integrate several key novel concepts". One of these concepts is team synchronization (which can be thought of timed coordination). STEAM framework is used to enable multiple team agents to have a representation of common goals, plans and joint commitments. It also provides several features to keep that representation consistent in a changing environment. These features are:

- team synchronization
- monitoring and repairing capabilities
- decision theoretic communication selectivity

Team synchronization is the ability of updating each agent’s state whenever its important. Let us imagine the following example, given by Tambe et al:

A squad of helicopters controlled by agents in a mission have to go from base to a point A, then wait the assigned scouts to tell them they can attack point B.

It is team synchronization that allows them to move all together (even if an agent processes the commands faster than the other), know if the scouts have reached the attack point and say its OK to advance or abort the mission if they all got shot down. One can say this can be achieved by heavily scripted behavior and normal synchronization, but that can only be done
CHAPTER 3. RELATED WORK

to a finite number of combinations in a very specific domain, and it is restricted to situations the developer thought. So if something unforeseen happens, the agents will not know what to do if they have a scripted behavior. That team synchronization can only be done effectively thanks to the monitoring and repairing capabilities of the STEAM framework. The framework allows the creation of team plans which contain the conditions for completion or failure of these plans. What happens is, every time a plan becomes unachievable (because one of its conditions become impossible to achieve), there is a call to the repair function, which then synchronizes every team member, possibly making them change their plans. Let us take the previous example: if all the scouts fail to report the attack plan will fail, therefore invoking a repair which will lead to another plan (either sending in more scouts or returning to base).

Because there can be many team plans and many agents together, STEAM also integrates selective communication, namely decision theoretic communication selectivity. This is based on communicating an agent’s belief when they terminate a joint intention only if the likelihood of that information being common knowledge is low. For instance, if a pilot is flying between waypoints he won’t be broadcasting his arrival to others, since they will probably know that eventually (considering that is the expected outcome), but he should broadcast if something unusual happens, like seeing an enemy tank patrol or being shot at.

As we can see, while Tambe et al had similar concerns, STEAM is a framework made to support flexible teams of NPCs. To do so STEAM relies heavily on team synchronization. The difference between synchronization and coordination however, is that synchronization is coordination regulated by a timekeeper (an instrument or person that measures the passage of time). This is easier to achieve in NPC-only teams where unambiguous communication and synchronization protocols are established beforehand, whereas a human player might not (and should not have to) follow a synchronous protocol of communication.

However communication is not all there is to coordination. Yen et al went further into it, giving much importance to anticipation. As they state in their work “efficient teamwork relies heavily on information sharing, especially in dynamic environments, but it must be done judiciously not to overwhelm the human participants with message passing”[10]. With that in mind they created CAST: a multi-agent architecture that relies on proactive communication as a way to reduce the cost on communication and create a more effective and flexible teamwork model [10, 9].

Flexibility and efficiency are two goals that conflict with each other. In order to have a flexi-
3.3. COORDINATION

In order to achieve satisfactory results in those two conflicting goals the authors try to create a \textit{shared mental model}\textsuperscript{1}. Since representing a full fledged \textit{shared mental model} is too complex and covers much more knowledge than needed in a particular problem, Yen et al\cite{9} chose to focus on two specific uses of it:

- improving teamwork through \textit{coordination} by anticipating actions and expectations. This is achieved through roles, capabilities and commitments.

- Using \textit{proactive information exchange} (asking for information when needed or giving it when it is deemed relevant, in other words, \textit{anticipating} the needs of information).

\textit{Proactive communication} increases effectiveness of communication in a number of ways since a) messages will only be sent when they are relevant, instead of being constantly sent; b) having information proactively sent reduces the need to ask for information and therefore reducing communication overhead; and c) even when the information needed is not proactively sent this mechanism can reduce the requests for information to a single provider, instead of all of them. In another paper \cite{10} they propose a decision-theoretic approach to cull the unnecessary communication. Why use a decision-theoretic approach? Because, when working together, team members need to deal with uncertainties since they only have partial information. The decision theory is a way of analyzing a series of strategies and choosing one when it is not known what is going to be the exact result of picking that strategy\textsuperscript{2}. This approach was adapted to the decision process of the information needer and provider, with emphasis on team-benefiting communication and provider/needer interaction.

Coordination and communication are clearly connected, but \textit{explicit} communication to achieve better coordination is not part of our goals on this work. We intend to create our agent capable of coordinating its actions with the human player without resorting to it. To do that we must rely on \textit{anticipation} mechanisms, which are interpretations of \textit{implicit} communication. An example of \textit{implicit} communication is observing the player movements. Through that, the agent will be capable of predicting the player future movement to a certain extent. If we recall the traffic example, although there is no teamwork, there certainly is coordination. \textit{This}

\textsuperscript{1}To know more about shared mental models, see \cite{15, 16, 17}

\textsuperscript{2}For more information about decision theory, see \cite{18}
happens because there is a shared set of rules (though drivers can actually communicate with each other through brake and directional lights). Both player and agent know exactly what they have to do and what they can do. Through that shared understanding we hope to achieve effective, though maybe not the most efficient, coordination. Using a set of communication mechanisms to improve coordination, cooperation and teamwork is an area for future work in this subject.

3.4 Motion Control

All the previous concepts converge to this section of our work. Teamwork, cooperation and coordination processes are meant to provide a set of actions to be executed. That execution is achieved through motion control. Although our agent operates in a virtual environment, research with reactive behavioral robot control is the area we find the most relevant for our work. While it has many concerns irrelevant to our problem (like sensor input processing from real world and physical interaction), their core issues are particularly aligned with this thesis primary goals. They need to react precisely and quickly to real-time events [30], have decision capabilities to deal with real world uncertainties [27] and often work in team environments [25] or even with humans [28]. Furthermore, there are many works regarding architectures and algorithms used to control these robots.

Before analyzing some architectures however, we shall start by looking at some methods to solve the major motion control challenge: moving around. Although the agent has a very concrete set of actions to interact with the environment, which include moving around, there is the need to know the exact sequence of actions necessary to reach the objective set by the decision mechanisms. One of the most popular algorithms to solve this type of issues is Rapidly-Exploring Random Trees (RRT) [26]. The concept behind the algorithm is growing a tree with all the possible moves within the space in hopes that one of the tree’s leafs will reach the goal. Although it has several techniques to ensure this search is performed rapidly, RRT only takes into account navigation, ignoring possible manipulation of the environment or even moving bodies.

A similar approach to this challenge is given by Zickler et al through Physics Based Planning [24]: using a physics engine to simulate future states of a possible interaction with the environment, it generates a tree of the possibilities in the hopes of achieving the desired state. they
3.4. MOTION CONTROL

also take into account moving bodies be it completely predictable, like an elevator, or partially predictable, like a human controlled body. This algorithm uses a reactive architecture called Skills, Tactics and Plays [27]. Skills are the actions the robot can do, Tactics are pre-programmed finite state machines of skills (see Fig. 3.2). Plays represent team-coordinated behavior, and are not used since this algorithm is related to single agent movement. With these skills and tactics, it is now possible to search through space with much more efficiency, since the physics engine will simulate the possible skill transitions from the current tactic instead of all the possible skills or impossible states. In order to simulate predictable bodies and partially predictable bodies, a set of reactive tactics and skills also have to be provided in order to simulate those bodies behavior. Even if they are not completely accurate, it is still preferable creating a rough tactic rather than assume the body is static. The planning algorithm is a loop that uses the active tactics in the current state in order to simulate new states in the physics engine. These new states are then added to the tree of simulated states, from which the next iteration of the algorithm will continue its search until a goal achieving state or the maximum number of iterations is reached.

A great advantage of this algorithm relies in the fact that it encases both path planning and goal oriented planning at once. Considering the fact that Geometry Friends is a game that relies in physics simulated interaction and is built around a physics engine also makes this a natural approach to our challenge. However, the original algorithm is based on an architecture that was not made for teams with human players, therefore having no consideration for issues such as coordination through anticipation. Furthermore, given our particular case, improvements could be made on the algorithm by adapting it to an architecture more suitable to our coordination and cooperation needs.

Figure 3.2: Example of a non-deterministic Tactic

Gat et al’s work [29], later referred in [28, 30], has some important considerations about planning and reacting. They defend the need for asynchronous, tiered solutions where com-
CHAPTER 3. RELATED WORK

Components with different behaviors according to their roles will often have different complexity and processing time needs. Therefore, an asynchronous architecture where computationally heavier components, like planning mechanisms, are processed independently from lighter components, like movement control algorithms, would greatly benefit not only a robot operating in a real world environment, but also an agent operating in a virtual world environment where synchronized actions are important and cannot be delayed or hindered.

Figure 3.3: 3T Architecture Diagram

Later works from authors such as Bonasso et al [28] further elaborated on this subject. They developed a tiered architecture whose main feature was the proved flexibility of their approach\(^3\). They also accounted for the possibility of the robot receiving guidance from a human supervisor, which is very relevant to our work. This architecture achieves robot control through three layers (hence the name 3T):

- A set of reactive skills coordinated by a skill manager. A skill can be viewed as an

\(^3\)this architecture was used in several different robot systems, all with very different hardware and functional characteristics
3.4. MOTION CONTROL

action that can be executed by the robot.

- A sequencer that activates and deactivates those skills according to the Reactive Action Package (RAP) active at a given time.

- A deliberative planner, that is able to reason about achieving specific goals given the current world model state and available RAPs.

A RAP is a package comprised of:

- One or more sets of tasks. Each task is a sequence of actions and preconditions.

- A goal that is achieved when any of the tasks in the RAP is executed successfully.

Whenever a certain RAP is activated, the RAP Interpreter will choose the task whose conditions better suit the current state of the World Model. An example RAP in a Robocup goalkeeper robot would be:

- **Goal** - Defend goal.

- **Task 1**
  - Precondition 1 - ball is in adversary possession
  - Precondition 2 - Has clear path to goal
  - Action - Intercept shot path

- **Task 2**
  - Precondition 1 - ball is in adversary possession
  - Action - intercept anticipated adversary move

- **Task 3**
  - Precondition 1 - ball is not in adversary possession
  - Action - do nothing

An action in a RAP can also be a RAP. This allows for increasingly complex actions. For instance, intercept anticipated adversary path could be a RAP with different tasks according to whether the adversary was surrounded or had a clear passing path.

In Fig. 3.3 it is possible to see the sequence in which objectives are turned into RAPs and subsequently into actions:
1. The Planner creates a tree of subgoals required to achieve the main goal he is set to. He will then go through each subgoal and activate their RAP one by one.

2. The active RAP is then read by the Interpreter, which will in turn determine which task from the RAP is the most adequate given the world model current state. It repeats the process on the task action if that action is a RAP, until it gets an action composed by only a set of skills, which is then sent to the Sequencer.

3. The Sequencer will execute that action through activation of the right set of skills. When an action is executed it either succeeds or fails, passing that information to the Sequencer. The Sequencer shall then pass onto the next action (in case of success) or deem the action a failure, reporting it to the Interpreter.

4. The Interpreter shall then try to activate a different task (checking the world model once again). When, and only when, the Interpreter exhausts all its options, the RAP is considered impossible to achieve, thus reporting that failure to the Planner, which has to decide whether to proceed to other objectives or re-evaluate its objective tree.

As the authors concluded, a task achieving robot (or in our case, an agent) can be realized with only the first or first and second layers (skills and sequencer), although it will have a scarce level of functionality. However this architecture deals with an uncontrolled agent such as a human operating with the robot in the same environment at the planner level, using counterplanning to reason about how conditions might be negated by uncontrolled actions. To overcome those uncontrolled actions, the planner includes extra RAPs to prevent the outcome of those actions. Using a RoboCup example, if there is a plan to pass the ball to another robot and there is an adversary nearby, the planner has to include a RAP for the passing itself and another RAP to deal with an adversary interception of the pass. In a highly cooperative scenario with an uncontrolled agent such as the human player, this poses the problem of having to create a large number of RAPs to cover all possible cases. This is unnecessary when one is using a physics based path planning approach.

Lensert et al also created an architecture that has been used effectively in RoboCup [25]. Although it is also a layered architecture, it differs greatly from Bonasso’s work. Since it was
created with the specific purpose of having a good performance in *RoboCup*, several key factors such as team coordination, low reaction times and positional awareness were very important for them. Their architecture had a modular behavior-based hierarchy used in conjunction with a combinator. By modular, they mean that each behavior could be added, modified or removed without affecting the whole architecture, and by hierarchical it means they could have different layers of behaviors operating at different levels of abstraction or scales of time, allowing for low reaction times as well as more complex (albeit slower) behaviors.

In order to keep their architecture like that, they used three hierarchies (Fig. 3.4):

- **sensor hierarchy** - within this hierarchy is stored all the information the robot has about the world. There is also an additional type of sensors, the virtual sensors. Virtual sensors use the regular sensor information in order to create more abstract information about the world (an example would be an image processing sensor that uses information from the camera on the robot, which is a sensor, to detect the position of the ball). These virtual sensors can also be stacked in several levels, in order to create a more abstract knowledge of the world (using the above example, one could create another virtual sensor using the ball detection sensor in order to create statistics of the ball position in the field).

- **behavior hierarchy** - Behaviors make all the choices for the robot. Different behavior
layers interact through controls (Fig. 3.4). They use the information from any sensor and controls coming from the above behavior layer in order to decide which controls to activate in the level below (or motor commands, in case the behavior is in the last level, behavior level 0). We can think of level 0 as motion control, level 1 as coordination, etc. This way, behaviors in higher layers are more complex as they will trigger a whole chain of lower level behaviors and controls.

- **control hierarchy** - Controls represent everything a robot can do. A control set behaves as a communication interface between two behavior levels. It represents information passed between those levels in data structures. They can also be seen as goals to achieve by the bottom layer (for example, a level 0 behavior can decide to activate a level 0 movement control instead of kick with the intent to have a level 1 clear path to goal control activated).

Another concept of extreme relevance to our work is the existence of non-conflicting behaviors: by using a combinator module, they can, at any given time, run more than one behavior at the same level of the hierarchy as long as they are non-conflicting. Non conflicting behaviors are behaviors that use different resources. Using a RoboCup example; a move behavior could be used at the same time as a look around behavior, since one uses the robot legs only, while the other uses its head. However none of these behaviors could be used together with a head butt behavior, which would need to control both the head and the legs in order to hit the ball correctly.

Of the considered most relevant features of this architecture (reactiveness, scalability and modularity), reactiveness is the one we value the most. We require low reaction times in motor control and normal sensors vs virtual sensors are a resourceful way of representing knowledge of the world and make it readily available to any modules. It is also possible to directly link the behavior levels approach to our view: we could easily map motion control, coordination, cooperation and teamwork into their own behavior layers.
4 Proposed Solution

After investigating how several researchers solved their motion control, coordination, cooperation and teamwork issues and identifying how these definitions correlate and overlap, we now present an agent framework that is capable of motion control, coordination and successfully identifies team cooperation points. It is also prepared to accommodate future work in the area of teamwork and cooperation. By looking into robot control we found out that many of our challenges have already been tackled in this field, but their solutions were not shaped to virtual game worlds where a human player is expecting teamwork and not just simple interaction, which is where most solutions fall short.

We will start by describing the agent routine, showing the steps taken by it to solve a level. A parallel will then be drawn from that onto the architecture tiers, their internal modules and necessary algorithms.

4.1 Agent Overview

![Figure 4.1: Example of a very simple level. W1 and W2 are nodes in the Navigational Graph, while C1 is the only Collectible, and thus objective, in the level.](image)

The best way to start describing our solution is by analyzing step by step how the agent solves a very simple level. The level chosen for this example is represented in figure 4.1. The first thing done by the agent in a level is to start perceiving its surroundings. To do so it uses
Sensors and a Navigational Graph that represents every place reachable by the agent or its teammate. In our example, this graph is represented by the black dots (W1 and W2).

All mechanisms used to perceive the game world are contained in the World Model layer of the framework (figure 4.3).

The agent then needs to know where to start. This is done by determining a sequence of objectives. In case of a Geometry Friends level, this would be the order in which to catch the collectibles. In this particular case CatchC1 would be the only objective, making the choice rather obvious.

After knowing what to do next, the agent needs to determine how to do it. This is done by choosing the most suitable Task given the circumstances. Since the collectible is not close enough to the square, it chooses a graph navigation Task, which we will call NavigateToC1.

In order to help visualizing this process, a sequence diagram was included in figure 4.2. A Task is a series of smaller steps connected to each other called AgentActions. The selected task from our example, NavigateToC1 is composed of two AgentActions: MoveToW1 and MoveToW2. When the task is selected, it will start the action MoveToW1, which will later transit to MoveToW2 if it executes correctly.

This AgentAction will then cause the agent to execute a certain predetermined sequence of actions for an amount of time. When the action is over, the agent checks whether it achieved its purpose or not and chooses another action accordingly. This is repeated until the Task completes (successfully or not). In either case, a new task is then chosen until the objective is achieved or deemed impossible to achieve. In our example, the agent checks if its position is nearby W1. If it is, it continues with action MoveToW2. If not, the current task, NavigateToC1, is considered over and the agent then proceeds to get a new one.

The agent would continue by choosing another task to grab the collectible, following the same process described for the NavigateToC1 task. That would make it complete the objective and the level.

This is the process followed by the agent in every level. Nevertheless, there are many details that were omitted from the step by step description for the sake of simplicity. These are described in detail in the following sections.
4.2 Agent Architecture and Modules

The layered architecture in fig. 4.3 was chosen because it allows the integration of the previously identified concepts in an autonomous way. Since each layer runs asynchronously they are somewhat independent and keeping the communication restricted to the neighboring layers also makes it easier to control concurrency. Although they are autonomous, it is common for a layer to wait for another layer’s output before continuing, as we show in section 4.2.5. The only common communication to all the layers is the World Model Layer, where all the information gathered about the agent’s world is obtained. The pieces of information gathered (called perceptions) are available to all the layers, so they can decide how to act based on it. Each of these layers will be explained in detail in the following sections.

Figure 4.2: Sequence diagram representing the agent steps in each layer when executing the level provided in figure 4.1
Figure 4.3: High Level Architecture Diagram. Representation of different layers and communication between them. Boxes next to bidirectional arrows represent data entities shared by both layers. The dark gray box represents the whole agent.

### 4.2.1 World Model

The *World Model* layer represents the agent knowledge of the surrounding world. This layer contains all the data about the state of the world, accessible by any other layer or its components. The components present in this layer are *Sensors*, *Virtual sensors* and the *Navigational Graph*.

Each *Sensor* executes asynchronously, simulating real world sensors. They contain the last valid state of information processed (e.g. there is a sensor in our agent that returns the actual square character position). This way it is possible to have access to valid data while the sensor is processing a newer state (even though this data might be slightly old, it should never be older than the update period of the *sensor*). A list of the sensors created for *Geometry*
Friends were:

- CirclePositionSensor
- SquarePositionSensor
- CircleVelocitySensor
- SquareVelocitySensor
- SquareHeightSensor
- CollectiblesPositionSensor
- IdleCircleSensor

Virtual Sensors work like regular sensors, but take their input from other sensors instead. An example of a created Virtual Sensor is the IdleCircleSensor which determines if the circle character has been idle for 5 seconds by comparing the current CirclePositionSensor value with previous values in the last 5 seconds.

In Geometry Friends we had our work simplified. Because the game uses a Physics Engine, Farseer Physics Engine, we just had to link the sensors to the relevant data, without having to calculate it (e.g. we could just access the Velocity field of an entity, instead of having to create a sensor that would calculate it based on the position variation at periodic time intervals).

While developing the solution, we reached a point where simple use of sensors as defined above was not enough to ensure correct level navigation. Hence, a Navigational Graph was necessary. This graph, composed by several NavNodes connected by NavPaths, represents possible paths each character can take from a point to another in the level. This graph generation process has many steps that have to be executed in a certain order. These are described in the following section.

4.2.2 Navigational Graph Generation

When we were developing our solution, we eventually got to a point where simply assessing the surroundings and acting accordingly was not enough to solve all the puzzles presented
in a level. For instance, we had issues determining whether or not a path existed to a cer-
tain location unless it was in a straight line. Figure 4.4 shows a case where the agent would
become stuck because he could not figure out how to reach the remaining collectible in the
level. The solution for this was to create a navigational graph (figure 4.5). The navigational
graph is a series of nodes connected by paths representing possible paths for each character.
There are different types of paths: normal (can be used by both characters), square-only (can
only be crossed by square), and circle-only (can only be crossed by circle). A very important
by-product of the navigational graph is the capacity to know what places are reachable by
both or any character. Thanks to that it is possible to determine whether a collectible is
achievable or not. This is a fundamental piece of the Cooperation Layer, as it provides the
planning modules a way to determine what is accessible by whom at any given position of the
level.
Figure 4.5: Level from figure 4.4 with first the version of *Navigational Graph*. White dots represent graph nodes. Lines represent a unidirectional path from the node they are touching to the node they are pointing to. Bidirectional paths are represented by lines on differing nodes pointing to each other. Please bear in mind that in closer nodes these lines overlap. Although they look like a single line, they still represent a bi-directional path between two nodes.
Our original node generation process was the following:

```plaintext
addNode(AgentStartingPosition);
addNode(PlayerStartingPosition);
generatePaths(PlayerStartingNode);
foreach Collectible in Level
    addNode(Collectible);
    generatePaths(CollectibleNode);
generateRegularNodes();
```

The process of node generation has to start from the player’s position, since its goal is to generate a graph that represents where each of them can reach. `generatePaths()` is only called for the `PlayerStartingNode` and not the `AgentStartingNode` because there needs to be at least 2 nodes in the graph in order to create a path (an origin and a destination).

The path generation occurs everytime a new node is added to the graph. This process, described in detail in section 4.2.2.2, checks for a possible path between every node already existent and the newly created one, hence the need to execute it after every creation.

After adding those two nodes, a node for each collectible is created. The set of nodes \{AgentStartingNode, PlayerStartingNode, CollectibleNode1, ..., CollectibleNodeN\} form the basis for the Normal Node Generation algorithm:

### 4.2.2.1 Regular Nodes Generation

This algorithm tries to create new nodes from the position of existent nodes. Because of this, it can only be executed after providing some starting nodes, which we do by creating a node in every character and collectible starting position. If any of these new possible nodes are not valid (by being placed too close to another node, or being inside or across an obstacle), it is discarded. If not, its paths are generated and the algorithm continues until no more nodes can be generated.

```plaintext
foreach Node in NodeList
    foreach PossibleNewNode in Node
        if isValid(PossibleNewNode)
            addNode(Collectible);
            generatePaths(CollectibleNode);
        else
            continue;
```
The result after running this process can be seen in figure 4.5. However we quickly realized a big caveat on this *Navigational Graph*: There were cases where two obstacles were placed close to each other with enough space for a character to pass, but the *Navigational Graph* would not identify it as a valid path because it could not generate a path to the *Nodes* on the other side. To overcome these situations, the node generation process was improved by adding a algorithm to place nodes aligned to each obstacle edges. This algorithm is processed before the *Regular Node Generation Algorithm*:

```plaintext
addNode(AgentStartingPosition);
addNode(PlayerStartingPosition);
generatePaths(PlayerStartingNode);
foreach Collectible in Level
    addNode(Collectible);
    generatePaths(CollectibleNode);
generateCornerNodes()
generateRegularNodes()
```

### 4.2.2.2 Corner Nodes Generation

![Corner Nodes Generation](image)

Figure 4.6: Again, level from figure 4.4, with the improved version of *Navigational Graph*. In this case there is no noticeable advantage in using the improved version.
This new algorithm generates possible nodes that are aligned to each obstacle’s edge (see figure 4.7). If these possible nodes are valid they are created and their paths generated.

Corner Generation Algorithm pseudocode:

```plaintext
foreach Obstacle in Level
    foreach Corner in Obstacle
        addNode(Corner);
        generatePaths(CornerNode);
```

It is important to note that both the algorithms have parameters to control the density of nodes and the maximum range of the paths. This keeps nodes from being created too close to each other, reducing the amount of redundant paths.

### 4.2.2.3 Generation of Paths Between Nodes

As we can observe in the previously presented algorithms, an execution of the path generation algorithm is mandatory after a new node is created. When analyzing the following pseudocode the reason for such limitation becomes clear:

```plaintext
foreach Node in Graph
    if (Node <> NewNode) AND Node.isInRangeOf(NewNode)
        if(pathIsBlockedByBlackObstacle(NewNode, Node))
            continue;
        else
            if(pathIsBlockedByGreenObstacle(NewNode, Node))
                tryToCreatePaths(NewNode, Node, squareOnly);
            else
                if(pathIsBlockedByYellowObstacle(NewNode, Node))
```
Figure 4.8: Example of a case where improved graph performs better than the original version. Notice how figure 4.8(a) has no link between the top and bottom, whereas figure 4.8(b) provides a path down.

\begin{verbatim}
tryToCreatePaths(NewNode, Node, circleOnly);
else
    tryToCreatePaths(NewNode, Node, both);
\end{verbatim}

All the possible nodes for the $NewNode$ are generated by checking if every existing node is within range of the new node. In case they are in range of each other, it is verified if no obstacle blocks the possible path. If a black obstacle blocks the path, no path is created. In case a colored obstacle blocks the path, the generation of a path with the corresponding agent restriction is attempted. It is important to mention this is only an attempt because, depending on maximum velocity requirements, a path may or may not be generated. A detailed description of this process is given on section 4.2.2.4.
Although it is possible to create a more efficient path generation algorithm (the current algorithm scans the whole set of nodes every time a new node is created, which is not very efficient), we did not deem it important as this algorithm is only ran when the level is loaded, which means there is no impact on agent performance.

In order to determine whether or not a path between two nodes exists, the algorithm checks the following:

- the height of the origin node
- the relative Y position of the destination node (in other words, whether the path direction is up or down)

If the path direction is up, a verification is made: if the vertical component of the path plus the origin’s node height is greater than the maximum height achievable the character, the path is not created for it. Since the agents have different movements, this means some paths are available for one of the agents while unavailable for the other. An example would be the yellow lines in figure 4.6. These represent circle-only paths. A square-only path would be marked in green, but there are no such paths in the aforementioned level.

However, if a path direction is down, no verification is made and it is automatically created. This happens because if, for some reason external to the agent capabilities, it happens to be in that node area, he will still be able to go down to any of the underlying nodes due to gravity.

This path selection is very important because it is what enables the agents to determine whether they can or cannot reach a place in the level. Although this solution works well in general, there are cases where, given enough velocity in the X axis, an agent could reach a place marked unachievable by the graph. An example of this limitation would be a large jump, like the one in figure 4.9.

### 4.2.2.4 Node Height Determination

The height of a node is determined by the distance from this node to the closest obstacle under it. A generic way to know the distance to the closest obstacle would be to run a line intersection algorithm\(^1\) between the vertical line starting on node position with the edges of

---

\(^1\)Paul Bourke has a great description on this algorithm in the following site: [http://paulbourke.net/geometry/lineline2d/](http://paulbourke.net/geometry/lineline2d/)
Figure 4.9: Example of graph limitation. There is no path available to the collectible, even though it is possible for the ball to jump from one platform to the other.

However, since obstacles in Geometry Friends are always rectangles aligned with both axis, it was possible to do a simplification of this algorithm:

```
height = maxint;
foreach Obstacle in Level
    if(NodePosition.X >= Obstacle.TopLeftCorner.X AND
       NodePosition.X <= Obstacle.TopRightCorner.X)
        /* note: (0,0) is the topleft point of the screen, so a positive distance means the obstacle is under the node.*/
        if(distance > 0 AND distance < height)
            height := distance;
```

While this simplification works in this specific case, it brings no performance gains to the agent during runtime because, like the previously mentioned algorithm, it is only executed
during level loading.

4.2.3 Pathfinding

The previous algorithms explain how the Navigational Graph is generated. This graph is then used by the agent in a multitude of ways:

- Used in Cooperation Layer to determine which collectibles are reachable.
- Used in Cooperation Layer to determine the sequence in which these collectibles ought to be caught.
- Used in Coordination Layer to initialize the Task that navigates waypoints.
- Used in Coordination Layer in certain tasks preconditions.

While the graph generation processes do not influence agent performance because they are ran at level loading time, the same cannot be said about pathfinding. Being a somewhat cumbersome process that is ran multiple times during execution, one must take care by not making it unnecessarily heavy. In order to keep it efficient, an A* search is used to determine the path. A squared euclidean distance heuristic is used instead of just the euclidean distance because we only need to compare distances and it spares the algorithm from square root calculations. The pathfinding algorithm method returns a list of nodes when the path is possible (empty list if the agent is already at its destination) or null if there is no path available. It also receives a parameter stating which game character the path is being calculated for (square or circle). This is rather important because, although the path returned for the human character might be impossible for the agent itself to use, the mere information of the possibility to reach a position (or lack thereof) is very important in a team context. It makes it possible for the agent to plan its sequence taking into account information about the capabilities of its teammate.

4.2.4 Teamwork

This is an empty layer representing future work to be made in Teamwork. It is included here to represent the fact that this solution was created with Teamwork in consideration. This layer is expected to support modules related to agent learning processes and/or personality
4.2. AGENT ARCHITECTURE AND MODULES

creation modules. These should, in turn, influence the way Cooperation layer determines its objectives. For instance, having a module determining whether the agent has a dominant or submissive personality would make it prioritize the individual objectives first and only then cooperating with the other player, while a submissive behavior would try to follow the other player and help him first.

4.2.5 Cooperation

This is where top level decision making processes are kept. Right now decisions are only made based on individual capabilities, but the agent is already capable of identifying some objectives achievable only through teamwork thanks to the pathfinding algorithm applied to the Navigational Graph. If we recall the example level (section 4.1), the first step after having an operational World Model was to define an objective sequence. This sequence cannot be just randomly chosen because catching the collectibles in the wrong order might render the level impossible to complete. Figure 4.10 is a clear example of a level where there are collectibles that must be caught before others.

4.2.5.1 Objectives sequence determination

To determine the order in which the collectibles must be caught, a simple algorithm is applied to a tuple representing the state of each collectible in the level. There are 4 possible states:

- **Possible Alone (PA)** - represents a collectible that, given the current position of an agent, is reachable by it.

- **Possible Team (PT)** - represents a collectible that, given the current position of the human character, is reachable by it.

- **Caught (C)** - represents a collectible that has been caught.

- **Impossible (I)** - represents a collectible that, given the current position of both characters, is unreachable.

Using the level in figure 4.10 as an example, the tuple would be:

if the agent went on and grabbed collectible 2, it would then look like this:

However, if it then went to grab collectible 3 instead of 1, the level would become impossible:
CHAPTER 4. PROPOSED SOLUTION

Figure 4.10: Example of a level with a specific order to catch the collectibles. For identification reasons, consider the first collectible the one with the top leftmost position.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>PA</td>
<td>PA</td>
<td>PA</td>
</tr>
</tbody>
</table>

The algorithm used to determine this sequence starts with the following tuple and expands it by catching each of the possible collectibles. Whenever it reaches a tuple where a previously possible collectible becomes impossible, it gives up on that path. If it reaches a tuple where all the collectibles are caught, it has reached a possible solution.

There is one improvement to this algorithm. There are equivalent tuples, i.e. situations where it does not matter which collectible A or B is caught, as long as both are caught before collectible C. In figure 4.10 we can see that collectible 1 and 2 can be caught in any order, as long as they are both caught before collectibles 3 and 4.

If we examine the tuples after catching either collectible 1 or 2, it is clear that only the state of the caught collectible changed. This is how equivalent collectibles are identified: when only the caught collectible state changes in tuples generated from the same original tuple, they are
4.2. AGENT ARCHITECTURE AND MODULES

Table 4.2: Tuple representing level with collectible 2 caught.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>C</td>
<td>PA</td>
<td>PA</td>
</tr>
</tbody>
</table>

Table 4.3: Tuple representing level with collectibles 2 and 3 caught with agent in a position where it is impossible to catch collectible 1.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>C</td>
<td>C</td>
<td>PA</td>
</tr>
</tbody>
</table>

considered equivalent tuples. Diagram 4.11 shows the tree of tuples generated and merged for the level in figure 4.10.

Figure 4.11: Diagram with a step by step representation of the objective sequence algorithm for level 4.10.

It is important to realize the limits of this algorithm. Although it is a very efficient way to determine the sequence in this scenario, especially if we consider the somewhat limited amount of collectibles a level has on average\(^2\), we assume that once a collectible becomes impossible to catch, this state is no longer revertible. This would not be the case if there were movable obstacles, since collectibles could be impossible to catch at a certain point in time and possible in another. This is a consideration that goes against the algorithm assumption

\(^2\)the average of all currently produced levels is around 6
that once a collectible is impossible, it will always be impossible.

After running this algorithm, the agent is left with an array of \( n \) integer elements, where \( n \) is the amount of collectibles in the level. Each element has a number representing the step in which that collectible has to be collected. Using the previous example, the array would be the following:

\[
\text{Table 4.5: Objectives sequence}
\begin{array}{cccc}
C1 & C2 & C3 & C4 \\
1 & 1 & 2 & 2 \\
\end{array}
\]

This array means collectible 1 and 2 must be collected on a first phase, followed by collectible 3 and 4 on a second phase. The agent always start at phase 1 and advances to following phases whenever there are no more collectibles to get on the current phase. Any impossible collectible will have value 0 and therefore be ignored by the agent. This allows for some flexibility when the agent is planning (if it has more than one collectible in a phase, it can choose to start by the closer one for example, or try to determine the shortest route through all the collectibles) while maintaining the necessary restrictions for the correct completion of the level.

Now that the agent knows the sequence in which the objectives have to be achieved, it is time to determine how to do them. This is done by selecting a task to be executed for a particular objective. Based on Firby’s work[8], we create a \textit{Reactive Action Package} (RAP) for every objective in the sequence. A RAP is an object that defines an objective and the actions necessary to accomplish such an objective:

- Goal - Objective this RAP will achieve in case one of its Tasks and all its SubRAPs are executed correctly.
4.2. AGENT ARCHITECTURE AND MODULES

- SubRAPs - Sub objectives that should be accomplished before attempting the current tasks. While there are sub objectives pending, no Task from this level will be executed.

- possibleTasks - A list of different Tasks that achieve this goal.

- State - State of this RAP. It can be SUCCESS, FAILURE or PENDING

4.2.6 Task Determination

![Figure 4.12: RAP entity details.](image)

Given a sequence, the agent then creates a RAP for each collectible. Although the framework allows for dynamic composition of RAP’s (by adding more tasks and/or subsequent RAP’s to an already created RAP), in *Geometry Friends* we ended up building a similar RAP for each collectible because the objective, to catch a collectible, was the same for all of them. Therefore, we created the following RAP:

- **RAP** *CatchCollectible N* composed by:
  - **Goal** - Is Collectible N caught?
  - **Sub RAP** *MoveToCollectible N* composed by:
    * **Goal** - Is Agent In Position?
    * **PossibleTask** - *NavigateToStaticPosition*
  - **PossibleTask** - *GrabCollectibleAlone*
  - **PossibleTask** - *GrabCollectibleTeam*

By checking the aforementioned list, it is possible to see how the *RAP CatchCollectible N* is composed of a *subRAP*; this behavior can be repeated indefinitely. A *RAP* will not execute its own *Tasks* until all its *subRAP* goals are successful. In this case, this means *CatchCollectible*
\textit{N} will only execute its \textit{Tasks} after \textit{MoveToCollectible N}'s goal is achieved. Whether the goal is achieved by executing the \textit{NavigateToStaticPosition} task or by any external factor, like being pushed by the teammate, is irrelevant.

When that happens and the agent is in correct position, it will go into the \textit{possibleTasks}. This is a list with a series of different tasks with the same outcome. In the \textit{CatchCollectible N} case, both \textit{GrabCollectibleAlone} and \textit{GrabCollectibleTeam} will provide the same outcome, which is to catch the \textit{Collectible N}. However, different \textit{Tasks} have different \textit{Preconditions}. The algorithm will traverse the list of possible tasks and will choose the first one whose \textit{ Preconditions} are satisfied. Hence, the order in which \textit{Tasks} are inserted in the \textit{RAP} will determine their level of priority. In this case, we can see the agent will always try to catch a collectible alone if the precondition allows it. Only when such is not possible will it try to do it through coordination with a teammate.

A \textit{Task} is comprised of a set of interconnected \textit{TaskStates}, forming a state machine, and a \textit{precondition}. The \textit{precondition} is a boolean expression that determines if the necessary conditions for its execution are met. If they are met, the precondition is valid and the task can be executed. For example, in order to execute the \textit{GrabCollectibleAlone} task, its precondition makes sure the collectible is in range of the agent’s character. If not, the algorithm will follow to the next task precondition and check if the collectible is in range of the combined character actions.

\section*{4.2.7 Coordination}

When the task is chosen, it is then sent to the \textit{Coordination} layer, where it is initialized. This initialization creates the task's state machine and signals the first \textit{TaskState} (figure 4.13) for execution.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{task_state.png}
\caption{TaskState content.}
\end{figure}
A TaskState represents a state in the state machine contained within a Task. It contains an AgentAction and has a Clause that is checked after its action is over. This Clause value determines which TaskState is chosen next. If it is true then nextStateIfSuccessful is triggered else nextStateIfFailure is triggered instead. However, if any of those states is null, an ActionCompleteEvent is triggered and the Cooperation layer chooses another one.

![State Machine for NavigateToC1 task](image)

Figure 4.14: State Machine for NavigateToC1 task. The value 1 represents a valid condition and the value 0 represents an invalid one.

By analyzing the task NavigateToC1 we can see that after starting it, the agent only proceeds to MoveToW2 if MoveToW1 is successful. This aborts the task in case W1 is not reached after MoveToW1 is executed. This triggers a task selection process in the Cooperation layer.

### 4.2.8 Motion Control layer

After being triggered in the Coordination layer, a TaskState sends its corresponding AgentAction to the Motion Control layer.

An AgentAction contains a timeline for each possible type of agent action. In the Square character case, there are two possible actions, Lateral Movement and Morphing. Therefore, every square AgentAction contains two Timelines. An example of this is given in figure 4.15.

![Graphical representation of the MoveToW1 action timelines](image)

Figure 4.15: Graphical representation of the MoveToW1 action timelines. Each bar represents a timeline. The values inside each segment are passed to the corresponding Actuator for their respective duration.
Timelines represent the actuator input at any given time in the AgentAction duration. In the example above, MoveToW1 action will have a duration of 450 milliseconds. The values inside the timeline represent the input passed to each action’s Actuator. Actuators in our solution have a similar purpose to their mechanical counterparts: they translate a certain input from the agent system into an action on the game world. Every Actuator in this framework is prepared to receive any value ranging from -1 to 1. However, the translation of this value onto an action varies from actuator to actuator. For example, the LateralMovement actuator converts any value below 0 onto a moveLeft command in the game engine, while a value larger than 0 is converted into a moveRight command\(^3\). If the value is exactly 0, the actuator activates nothing. If we look at the first timeline from the example above, we can now understand the different values in it. The -1 value in the first segment means the actuator will activate the moveLeft command for 350 milliseconds since the action starts, followed by idling for 50 milliseconds and finally activating the moveRight for 100 milliseconds. During all this time the Morph timeline has the value 0, meaning its actuator will stay idle for the whole action duration.

4.2.8.1 Predicting an action result

Why are there so many steps in a straightforward action such as a simple lateral movement? The answer is: Physics. Geometry Friends has a Physics Engine that simulates friction, inertia, torque, etc. Furthermore, all of characters movements are applied by a force, not a linear velocity. This means that after applying a move command in any direction, the character will continue moving a bit longer until friction makes it stop. All these factors, coupled with collisions, make it very hard to predict precisely the result of such actions.

Several different types of actions were tried, like simply applying a moveLeft command followed by a long enough period of idleness to let the character stop by itself, but that was rather inefficient, not to mention it would also be quite unpredictable since the amount of friction depends how much of the character is in contact with any surface at any given time. We ended up applying moveLeft for a longer period and countering it with a force in the opposite direction, through the moveRight command, to make brake it. Although using the latter approach was indeed much more efficient, it was found during experimentation that abruptly

\(^3\)These commands in the game engine are what actually make the game character move. They are the same commands executed when a human player presses the keyboard while controlling the character. Therefore, there is no difference for the game when an agent controls the character instead of a human player.
transiting between forces in opposite directions without allowing a small period of idleness would cause the character to lose balance, especially when stretched, as if it tumbled in a small, invisible obstacle. For that reason we added the 50 milliseconds idle time between the -1 and 1 segments in the timeline.

Since this layer is in direct control over the agent actions, it is the one with the fastest execution cycle to keep agent reaction times to a minimum. Therefore, this layer’s code is executed every frame which, in optimal conditions, means it is executed 100 times per second\(^4\). This means that there are 10 milliseconds to calculate everything needed to create one frame in the game. This includes the agent processes, the Physics Engine simulations, the Game’s Logic processes as well as Graphic’s Engine calculations. For this reason, processing done in this layer is kept simple. Processes at this level are limited to choosing the correct input for each Actuator based on a current AgentAction and the time passed since this AgentAction was activated.

4.3 Testing Tools

Creating each of these layers of abstraction on top of each other was a gradual process. By starting from the bottom layers and progressing through the architecture, it was possible to test each one before progressing to the next level. However, there are always issues that only surface on later stages. For instance, bugs due to concurrent threads trying to access the same resource did not appear while we were working only on Motion Control layer, since there was only one thread accessing to all the resources.

It was important to create tools to facilitate debug and help visualize the rationale behind the agent actions. While logging all steps was something done from the beginning, it soon became a cumbersome task to read such logs. Because of that three major tools were created in order to help the testing and debugging process: the Level Editor, Graph Viewer and Agent Data Viewer.

4.3.1 Level Editor

The Level Editor is not a debugging tool, but it was an invaluable testing tool. Being able to create, load and edit any level from the game and test it right away was surely faster than

---

\(^4\)the game engine is limited to a maximum of 100 frames per second.
having to edit the XML files by hand. Because it had no direct relation to the thesis work, it was also the ideal tool to learn by experimentation how GeometrY Friends was programmed and structured. Hence, the Level Editor appeared before the agent itself. As can be seen on figure 4.16, it follows a basic toolbox metaphor: a bar on the left side of the screen contains the possible tools that can be used to place objects on the level area. When one tool is selected, a brief description and its name appear on the top of the screen and a different cursor can be dragged in the grid in order to create a shape.

Whenever the Save & Exit button is pressed, the user is asked if he wishes to save or not. Also, it checks if the level is valid. A valid level must satisfy all of the following conditions:

- There must be a Circle Character starting position
4.3 TESTING TOOLS

- There must be a Square Character starting position
- There must be at least one Purple Collectible

Once all these conditions are satisfied, the level can be saved and played in Multiplayer or Singleplayer mode.

All the levels used to test the agent were created using this tool.

4.3.2 Graph Viewer

The graph viewer only appeared in the later stages of the agent development, along with the agent data viewer. The need for this tool appeared nearly as soon as it was realized a navigational graph was necessary. Since there were certain nuances in the whole path and node generation process, a tool for correct visualization of this graph, along with certain information about its components was very important.

This tool is composed by 2 parts: the graph viewer itself and the path viewer. The graph viewer is the tool used to see all the graph nodes and paths and was used in figures 4.5, 4.6, 4.8(a), 4.8(b) and 4.9. It also has a mode where it shows either each node coordinates or each path position requirements. These modes are toggled by pressing the TAB key. The path viewer can be activated at any time during runtime by pressing a number from 1 to 9. It automatically shows the path between the agent and collectible 1 to 9 respectively, as long as the path or collectible exists.

4.3.3 Agent Data Viewer

The Agent Data Viewer (figure 4.18) is a panel that can be turned on during the game in order to see which RAP, task and action is active at the agent. Since the actions and tasks can change rather quickly, a quick fading animation is played whenever the display data changes. This provides an easy way to determine whether a certain action or task is being updated too fast or not being updated at all. This panel can be activated by pressing the F1 key.

This information could be traced with logs, but it was a cumbersome process and it was rather difficult to do in real time. With this tool, debugging certain unexpected agent behaviors became much easier.
4.4 Evaluation

In order to test and evaluate our agent we modified Geometry Friends by creating a Single Player mode where the NPC can control either character. We also made the necessary modifications in the game to support two agents playing with each other. This made it possible for us to see how easy it is to adapt this solution to a different agent context. There was no need to modify the World Model in both agents at all because the world is exactly the same. However, we did have to create new Actuators, AgentActions and Tasks, since all of these components are inherently tied to the unique movements of each character. However, work in the circle actions is considerably less elaborate. For instance, there is no prediction on where the circle will land after a jump, or even what its trajectory is going to be. This makes the circle miss many of its jumps. There were a series of challenges specifically related to the circle character motion control that could not be solved in time for this thesis. These are described in detail in section 6.

The test levels were created while developing the agent. Levels like 4.19 and 4.20 were created before the agent could even move. These were used as a baseline during the whole process, as the bare minimum motion control required to deem the agent acceptable. By the time the
agent could catch the first two collectibles in level 4.20, we realized a navigational graph was essential.\footnote{Videos of the agent performing some of these tests can be seen in http://geometryfriends.tumblr.com.}

The evaluation process consisted on presenting the agent with varied challenges and situations. By observing how it would overcome them or fail trying, several new mechanisms were created. For instance, during the first iterations there was no objective sequence determination (although it was already identified as an issue, as we can see in section 4.2.5.1, a solution was not ready by that time). However the agent was able to complete the level in figure 4.20 as soon as the navigational graph was finished, because the collectibles were added to the level in the correct order and the agent would solve them by that order. By creating a level where this order was specifically wrong (figure 4.23) we were able to test the objective sequence determination algorithm and find out other problems in Motion Control issues such as movement prediction.

Each referred level was used as a testing ground for at least one agent challenge, providing important insight on the agent behavior and allowing the identification of issues that were not initially predicted. We will now present these levels, describing why they were designed, which challenges they provided to the agent and how these were or were not solved. We will also compare the agent time with the average beginner and expert player time. The average beginner time was recorded during a public demonstration session of the game. During this session anyone could come and play the game. The amount of data varied from level to level since not every team tried all the levels. There were however more than 10 plays on each level with data. The best 10 times were taken and averaged in order to provide the numbers presented below. The expert player time was an average of the best 5 scores made by two of the game developers.

### 4.4.1 Test Level 1

This was actually the first level of the original game (figure 4.19). It was used during first phases of development, while only the motion control layer and coordination layer were being tested. It was possible to make the agent execute a coordinated action with the human controlled circle. However this level is now impossible to complete with the agents because the graph cannot generate a path to it. This problem is explained throughly in section 6. Since this is a level that cannot be completed by two agents, there is no point comparing the
4.4.2 Test Level 2

This level (figure 4.20) was the first created with the sole purpose of testing the agent. As can be observed, the circle is kept in a closed area not only to make sure only the square moves around, but also to test how the graph generation would deal with the situation. The main challenge here was the pathfinding. The solution was provided by creating the navigational graph.
4.4. EVALUATION

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 seconds</td>
<td>12 seconds</td>
<td>13 seconds</td>
</tr>
</tbody>
</table>

Figure 4.21: Impossible level for square character

4.4.3 Test Level 3

This is an impossible level for both agent and human players controlling the square character (figure 4.21). It was designed to make sure the agent would complete everything it process impossible objectives and ignore them accordingly. Although the level is possible if the circle tried to reach it, our implementation could not conclude it.

4.4.4 Test Level 4

Figure 4.22: Level with tight vertical passage
CHAPTER 4. PROPOSED SOLUTION

This level (figure 4.22) served not only to test specific cases in the graph as it also served to test the square agent preciseness in motion control. It can be seen in the time score that the expert player is much faster than the agent. However, this is hard to accomplish by novice players, who take nearly as much time as the agent.

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>45 seconds</td>
<td>16 seconds</td>
<td>53 seconds</td>
</tr>
</tbody>
</table>

4.4.5 Test Level 5

In this level (figure 4.23) both the objective sequence determination algorithm and the capacity to cross small horizontal gaps were tested. In this level the agent was slightly slower than the expert player, but faster than the beginner.

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 seconds</td>
<td>12 seconds</td>
<td>13 seconds</td>
</tr>
</tbody>
</table>

4.4.6 Test Level 6

This level was made specifically to test the objective sequence determination (figure 4.4.6). It is possible to observe how slower the agent is versus any of its human counterparts. This happens because it does not optimize the order in which to grab the collectibles, making it take longer.
4.4. EVALUATION

Level with parallel objectives in sequence and fine motion control

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 seconds</td>
<td>9 seconds</td>
<td>23 seconds</td>
</tr>
</tbody>
</table>

4.4.7 Test Level 7

In this level (figure 4.24) the square agent has to morph up and down in order to get one collectible and reach the other. The purpose of this level was to test both the agent movement and the graph generation. In this case the agent outperformed both human players averages.
4.4.8 Test Level 8

This level (figure 4.25) was designed to explicitly observe two agents playing in the same area and see whether or not they compete for the same collectibles and how they reacted to the collisions. We were also able to test some basic jumping actions in the circle agent. From this level onwards there is no data from human players since these were created after the human player tests took place.

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>4 seconds</td>
</tr>
</tbody>
</table>

4.4.9 Test Level 9

This level (figure 4.26) had a slightly harder jump, which proved to be too complex to be accurately predicted by the circle agent. Although they could finish it, it was possible to observe how limited current circle agent actions were.

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>32 seconds</td>
</tr>
</tbody>
</table>
4.4. EVALUATION

4.4.10 Test Level 10

With this level (figure 4.27) we discovered an error in our navigational graph generation algorithm. Although there is an obstacle blocking the access of the square character to the collectible on the right, the graph informs the agent a path is possible. It makes the agent continuously try to grab that collectible until the circle finally reaches it\(^6\).

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^6\)It is possible to watch a recording of this behavior in [http://geometryfriends.tumblr.com](http://geometryfriends.tumblr.com)
4.4.11 Test Level 11

Figure 4.28: Another level with a graph limitation. Graph can be seen in figure 4.9

This is another level (figure 4.28) created to demonstrate another limit in the navigational graph. Although it is possible for the circle to jump from the left platform to the right platform, the graph deems it impossible. This happens due to the way paths are being created. Since paths between nodes just depend on their height, high platform jumps are considered impossible.

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Expert</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
5 Conclusion

Teamwork between agents is a relatively recent area of research, with its first articles released in the early 90’s. The subject of human - agent interaction as team members is even more recent: most of the work (if not all) about it has only been published in the latest decade. If we consider how new this area of expertise is, it is only natural that there is still a lot of ground to be covered. However, motion control issues have been analyzed for a much longer time in robotics control. For this reason our solution draws much of its inspiration from this area. However, the focus of our work was to provide a framework in which teamwork, cooperation, coordination, motion control could interweave efficiently. Therefore, by keeping each of these key concerns in mind and tapping from the work done in robotics control, we were able to assemble a solid foundation in terms of motion control and coordination while demonstrating how cooperation can be achieved through several mechanics and still provide room for improvement in cooperation and teamwork areas available for future work such as joint intentions, decision making capabilities, social awareness, human - agent communication modules or even learning mechanisms.

Although limitations to our solution’s implementation are identified throughout this document, we believe this thesis goal was attained successfully. Many paths for future work based on this framework are now open, and it would be very rewarding to see this agent evolve into a complete synthetic character.
6 Future Work

As mentioned previously, although this thesis objective is to provide an agent with motion control and coordination mechanisms in order to solve challenges in a virtual world and identify cooperation points, great care was taken during the whole process to ensure the solution would be a flexible and easily adaptable framework capable of accommodating more features. Hence, several new issues and research directions were revealed during the development. In this chapter we are going to start by enumerating limitations related to motion control that we identified and could not improve in this work, either by time constraints or relatively low relevance to the thesis main topic. We will then proceed to provide some suggestions on future features and research directions.

6.1 Identified Improvements

As referred in section 4.4, the Navigational Graph became a core piece of the agent, but it was only identified as necessary later in the project. Furthermore, modifications to its generation must be done with great attention to detail in order to keep previous functionality unscathed. For that reason several shortcomings on this graph were identified, but impossible to correct in time for this delivery.

Besides the limitations detected on levels 4.27 and 4.28 on section 4.4, there are also improvements to be made in the graph in order to allow it to fully identify all the places reachable by the team. There is also the issue of circle jumps. More often than not, the circle character has to jump onto platforms. These jumps might require moving in the opposite direction of the destination in order to gain enough velocity for it to work. Currently there are no mechanisms to detect that, which means the circle can only do simple jumps. There are no way to determine the reach of each character as a team, i.e., it is possible to determine where the circle or square can reach by its own, but there is no consideration on team moves. Hence, positions accessible by simple teamwork are not tried because the agents rely on the graph
to tell them where they can reach, and the graph deems it impossible. A possible solution to these limitations would be creating another type of path between nodes representing team interactions, as well as path minimum velocity requirements on each of them, along with actions on the agents to control these velocities.

Although the agent does not have many heavy processing algorithms in this phase, it would be naive to think that with future work that will stay that way. Therefore another good measure would be to create some sort of cache for the graph, allowing to save several pathfinding operations. This presumes the level will not be constantly changing and making these previously calculated paths obsolete. That is another consideration for future work. By adding dynamic obstacles to the game, several mechanisms and simplifications on the current algorithms become void. A good example of this is the node height determination algorithm, or even the node generation algorithms.

6.2 Suggested Features

The areas where the agent is lacking most are related to Teamwork and Cooperation. This is natural, since this thesis goal was to provide a good foundation for future work in these areas. However, one possible future feature that would be useful is a module in Motion Control layer with capacity to create very fast reactions to certain events, imitating humans ‘instinctive reactions’, e.g. stopping right before colliding with an obstacle or character, even when an action tells it to go in that direction. These ‘instincts’ would have to provide a way to be overridden by specific actions. An example would be a reaction to avoid falling from a platform, and a jumping down action that would have to specifically override that reaction. This would make the agent behave more humanly as well as greatly improve its motion control response to unpredictable events.

This thesis focused on coordination without explicit communication. Nevertheless, there is an undeniable correlation between communication and teamwork. One of the common denominators observed in the two human players play-tests was the quantity of verbal, and sometimes physical, communication between both players. From discussing the order in which they were going to do the level and coordinating movements to taunting and encouraging each other, constant communication was maintained. The existence of a communication mechanism be-
6.2. **SUGGESTED FEATURES**

tween human and agent would be a very good addition to the agent capabilities. It would be a good way to convey its intentions to the human player, to ask for help acting and deciding and also make it seem more human.

The ultimate goal for such agent would be to able to *learn from the player*. Such possibility would make it capable of gradually adapting its own play style to a player, improving their efficiency as a team by each play session together. In order to do such thing, some adaptations would have to be made to the current solution, such as a way to save and load this acquired load into a file or database. As it is, the agent does not store any information between levels. Having some sort of *Personality* simulation is also a possible research direction somewhat related to the previous suggestion. It would be interesting to simulate different types of personalities in game and observe how these personalities complement of clash with each other. This would then be tested through human-agent interaction or agent-agent interaction.
Bibliography


