

**FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO**

# **Humanoid Robotic Soccer Realistic Simulation**

**Igor Silveira**



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Luís Paulo Reis

July 31, 2020



# **Humanoid Robotic Soccer Realistic Simulation**

**Igor Silveira**

Mestrado Integrado em Engenharia Informática e Computação

July 31, 2020





# Resumo

Os conhecimentos nas áreas da robótica e da inteligência artificial têm aumentado substancialmente na última década. Uma das principais contribuições do lado não lucrativo de pesquisa e desenvolvimento é a iniciativa RoboCup, que desafia investigadores de todo o mundo a continuar a desenvolver e melhorar as tecnologias através de vários desafios organizados, sendo um deles a Liga 3D de Simulação de Futebol, visualizada através do monitor RoboViz.

O trabalho desenvolvido nesta tese foca-se no desenvolvimento de uma extensão do visualizador RoboViz que suporta o cálculo e a detecção de eventos em tempo real da simulação, fornecendo feedback relevante aos investigadores envolvidos, além de uma experiência de visualização realista para os espectadores com base numa arquitetura orientada a eventos.

A solução desenvolvida é capaz de renderizar informações estruturadas na tela acerca da partida atual e possui um novo modo de câmara que se assemelha a uma transmissão na vida real, proporcionando uma sensação mais realista comparando com o sistema atual. A sua arquitetura modular permite um fácil desenvolvimento sobre a versão lançada por este trabalho, permitindo uma introdução simples de novos cálculos e novos lançadores de eventos.

Os resultados obtidos, através de questionários a população em geral, confirmam uma melhor leitura do fluxo do jogo e do desempenho da equipa e também uma experiência de visualização melhorada ao comparar com o visualizador original. Portanto, podemos concluir que a coleção de dados em tempo real é crucial para entender o fluxo do jogo e obter uma melhor noção do desempenho da equipa.

**Palavras-Chave:** Agentes, Sistemas Multi-agente, Futebol Robótico, RoboCup, Computação Gráfica, Cinematografia, Câmara Virtual, Transmissão em Direto, Orientada a Eventos

**Email:** igorasilveira@gmail.com



# Abstract

The fields of Robotics and Artificial Intelligence have been soaring over the last decade. A main contributor on the non-profitable R&D side is the RoboCup initiative which challenges researchers from all over the world to keep developing and improving the technologies through several distinct organized challenges, one of them being the RoboCup Soccer Simulation 3D League which is visualized through the RoboViz 3D monitor.

This thesis work focus on the development of an extension of the RoboViz visualizer that supports the calculation and detection of real-time events of the simulation providing valuable feedback to the researchers involved as well as a realistic visualization experience for the spectators based on an event-driven approach.

The developed solution is able to render structured information on screen about the current match and has a new camera mode that resembles a real-life directed broadcast giving it a more realistic feel over the current system. Its modular architecture allows for a facilitated further development on top of the version released by this work enabling a simple introduction of new calculations and event dispatchers.

The results obtained, through general population questionnaires, confirm a better reading of the game flow and team's performance and also an enhanced visualization experience when comparing with the original visualizer. Therefore, we can conclude that real-time data collection is crucial for understanding the game flow and better notion of team performance.

**Keywords:** Agents, Multi-Agent Systems, Robotic Soccer, RoboCup, Graphical Computation, Cinematography, Virtual Camera, Live Broadcasting, Event-Driven

**Email:** igorasilveira@gmail.com



# Acknowledgments

This section is intended to acknowledge all those who made it possible to get to this stage of my education and those who had a direct influence in the development of this thesis.

I'd like to thank Prof. Dr. Luís Paulo Reis for his orientation on the development of this work along with his suggestions and recommendations pushing me to deliver my best possible work. I would also like to thank Prof. Dr. Nuno Lau for his valuable inputs and contributions that aided in the development of the project. And last but not least, Prof. Brigida Mónica Faria for her formidable guidance with the results analysis.

Also a tremendous thank you to all my friends and family but more specifically to my mother, Esperança Amorim, and my aunts, Dina Pereira and Cidália Silveira, whom without none of this would come to be. Despite many problems and difficulties, they enabled me to achieve this accomplishment.

Last but not least, my biggest show of appreciation goes to my "other half", Ana. Thank you for always being there, for pushing me to always do my best and for enduring these tough times of much work and little time. The final word goes to my four legged friends, Luna and Açúcar, that were also always present to show much love and support.

To all, thank you.

Igor Silveira



*“Our acts can be no wiser than our thoughts.  
Our thinking can be no wiser than our understanding.”*

George S. Clason





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context and Motivation . . . . .	1
1.2	Objectives and Approach . . . . .	2
1.2.1	Thesis Question . . . . .	2
1.2.2	Approach . . . . .	2
1.3	Structure of the Dissertation . . . . .	3
<b>2</b>	<b>Autonomous Agents and Multi Agent Systems</b>	<b>5</b>
2.1	Intelligent Agents . . . . .	6
2.1.1	Agent Definition . . . . .	6
2.1.2	Agent Programs . . . . .	7
2.1.3	Agent Autonomy . . . . .	9
2.1.4	Agent Environment . . . . .	10
2.1.5	Agent Architecture . . . . .	11
2.2	MAS - Multi-Agent Systems . . . . .	17
2.2.1	Distributed Problem Solving vs Multi-agent systems . . . . .	18
2.2.2	Multi-agent Systems Motivation . . . . .	18
2.2.3	Agent Communication . . . . .	19
2.3	Conclusions . . . . .	26
<b>3</b>	<b>Graphical Computation</b>	<b>27</b>
3.1	Brief History . . . . .	27
3.2	Pixel - graphics base unit . . . . .	28
3.3	OpenGL - 3D Graphics API . . . . .	29
3.3.1	Primitives . . . . .	30
3.3.2	3D Coordinates . . . . .	30
3.3.3	3D Transforms . . . . .	31
3.3.4	Hierarchical Modeling . . . . .	34
3.3.5	Projection and Viewing . . . . .	36
3.3.6	3D Visualization Pipeline . . . . .	38
3.4	Conclusions . . . . .	39
<b>4</b>	<b>Cinematography: The Art of Visual Storytelling</b>	<b>41</b>
4.1	The Principles of Cinematography . . . . .	41
4.2	The Camera . . . . .	42
4.2.1	Virtual Camera vs Real Camera . . . . .	42
4.2.2	Physical Restrictions of the Camera . . . . .	43
4.2.3	Intelligent Camera Control . . . . .	43

4.3	Conclusions . . . . .	44
<b>5</b>	<b>Domain Application Analysis: Human and Robotic Soccer</b>	<b>45</b>
5.1	Human Soccer . . . . .	46
5.1.1	History . . . . .	46
5.1.2	Characteristics . . . . .	48
5.1.3	Tactical Evolution . . . . .	49
5.2	Robotic Soccer - RoboCup . . . . .	49
5.2.1	RoboCup Leagues . . . . .	50
5.3	The RoboCup Soccer Simulator . . . . .	58
5.3.1	SimSpark . . . . .	58
5.3.2	Spark . . . . .	59
5.3.3	SoccerServer . . . . .	60
5.3.4	RoboViz . . . . .	61
5.3.5	Log Player . . . . .	63
5.4	Conclusions . . . . .	63
<b>6</b>	<b>Project - LiveDirector for RoboViz</b>	<b>65</b>
6.1	Architecture . . . . .	65
6.1.1	Configuration File System . . . . .	66
6.1.2	BallEstimator Class . . . . .	67
6.1.3	StatisticsParserListener Interface . . . . .	69
6.1.4	StatisticsParser Class . . . . .	70
6.1.5	StatisticsOverlay Class . . . . .	81
6.1.6	LiveDirectorCamera Class . . . . .	91
6.2	Conclusions . . . . .	96
<b>7</b>	<b>Result Validation Analysis</b>	<b>97</b>
7.1	Survey . . . . .	97
7.1.1	Participant Characterization Questions . . . . .	97
7.1.2	Classification Questions . . . . .	97
7.2	Analysis . . . . .	101
7.3	Spearman's Rank-Order Correlation . . . . .	103
<b>8</b>	<b>Conclusions and Future Work</b>	<b>105</b>
8.1	Future Work . . . . .	105
	<b>References</b>	<b>107</b>

# List of Figures

2.1	An agent in its environment (adapted from [41]). The agent takes sensory input from the environment and produces, as output, actions that affect it. [55]	6
2.2	Basic Architecture of an Autonomous Agent (adapted from [32])	12
2.3	A goal-based agent architecture where it has explicit goals (adapted from [41]).	13
2.4	An utility-based agent architecture where it maps a given state to an actionable measurement (adapted from [41]).	14
2.5	Actions selection in a layered subsumption architecture (adapted from [36])	15
2.6	Information and control flows in three types of layered agent architecture (adapted from [55])	16
2.7	A BDI Agent Architecture (from [12])	16
2.8	Basic Architecture of a Multi-agent system (from [55])	17
2.9	(a) Multi Agent System (b) Distributed Problem Solving (from [12])	18
2.10	A federated system architecture (from [19])	20
2.11	The design space of agent communication languages. The region in the left represents existing ACLs, which follows a mental agency model. The region in the upper right represents the desired goals, which dictate a social agency model (adapted from [35]).	25
3.1	Ivan Sutherland using Sketchpad in 1962 (from [24])	28
3.2	Example of line primitives behaviour in OpenGL (from [13])	31
3.3	Example of triangle primitives behaviour in OpenGL (from [13])	31
3.4	Example of quads primitives behaviour in OpenGL (from [13])	31
3.5	Differences in Axis orientations in both Blender and OpenGL (from [39])	32
3.6	Example after rendering two faces (from [13])	36
3.7	Viewing transformation process (from [13])	37
3.8	Possible Virtual Camera Movements (from [33])	37
3.9	Virtual 3D Camera Parameters (from [33])	38
5.1	Soccer popularity around the world (from [1])	45
5.2	<i>Kemari</i> illustration by Akisato Rito 1799 (from [31])	46
5.3	Stone carving that shows a man balancing a ball (from [3])	47
5.4	a) First International: Scotland 0 England 0; b) Wrexham 1 Druids 0; c) Uruguay 4 Argentina 2 (from [53])	50
5.5	RoboCup Initiatives Structure	51
5.6	2D Simulation League (from the official website)	52
5.7	3D Simulation League (from the official website)	52
5.8	The humanoid NAO robot that was modeled for the 3D Leagues	52
5.9	Small-size League game (from the official website [2])	53

5.10	Middle-size League game (from the official website [2]) . . . . .	54
5.11	Standard Platform League game (from the official website [2]) . . . . .	54
5.12	Humanoid League game (from the official website [2]) . . . . .	55
5.13	Rescue Robot League game (from the official website [2]) . . . . .	55
5.14	Rescue Simulation League simulation (from the official website [2]) . . . . .	56
5.15	RoboCup Industry @Work (from the official website [2]) . . . . .	57
5.16	RoboCup Industry Logistics (from the official website [2]) . . . . .	57
5.17	Junior Soccer League (from the official website [2]) . . . . .	58
5.18	Junior OnStage League (from the official website [2]) . . . . .	58
5.19	SimSpark - a multiagent generic simulator . . . . .	59
5.20	Spark Architecture (from [9]) . . . . .	59
5.21	SoccerServer Architecture (from [12]) . . . . .	60
5.22	Architecture for RoboViz, SimSpark server (rcssserver3d), agent, and user inter- action (from [46]) . . . . .	62
5.23	RoboCup Soccer Simulation 3D Match in RoboViz . . . . .	62
6.1	Observer pattern implementation . . . . .	66
6.2	<i>Statistic</i> class and <i>StatisticType</i> enumerable definition . . . . .	67
6.3	Diagram of the forces acting on the moving ball . . . . .	68
6.4	Validation of the model results for position estimation . . . . .	69
6.5	Representation of the migrated BallEstimator Class . . . . .	69
6.6	Representation of the <i>StatisticsParserListener</i> Interface . . . . .	70
6.7	Representation of the statistics <i>Map</i> during a match . . . . .	71
6.8	Message atomic structure representation . . . . .	72
6.9	UML representing the <i>StatisticsParser</i> class hierarchy . . . . .	72
6.10	Representation of the structured message for an offside for the left team . . . . .	73
6.11	Representation of the class's <i>Map</i> after addition of the offside message . . . . .	74
6.12	Representation of the structured message for a crowding foul by left team's player 9 . . . . .	74
6.13	Representation of the class's <i>Map</i> after addition of the foul message . . . . .	75
6.14	Possession scores for given values example . . . . .	77
6.15	Representation of the field's coordinate system . . . . .	77
6.16	Representation of the initially empty position matrix . . . . .	78
6.17	Representation of the attacking zones of the right team . . . . .	79
6.18	Representation of the case of an <i>shot on-target</i> that ends outside in the <i>off-target</i> y-coordinate space . . . . .	80
6.19	Representation of the possible shooting scenarios . . . . .	81
6.20	Menu item for toggling the Statistics overlay . . . . .	82
6.21	Representation of the <i>StatisticsPanel</i> class . . . . .	82
6.22	Representation of the <i>StatisticsPanel</i> class . . . . .	83
6.23	Representation of the <i>StatisticsPanel</i> class's <i>Map</i> of panel times . . . . .	83
6.24	Representation of the <i>StatisticsPanel</i> class's <i>Map</i> of panel times after a few sec- onds of gameplay . . . . .	84
6.25	Timeline gadget on a running game with goals . . . . .	84
6.26	Representation of a calculated occupation percentage matrix . . . . .	85
6.27	Heat Map output at the final stages of a match . . . . .	86
6.28	Foul panel at a late stage of a match . . . . .	89
6.29	Possession graph at a late stage of a match . . . . .	90
6.30	Possession graph at render stage 1 . . . . .	90
6.31	Possession graph polygon calculation illustration . . . . .	91

6.32	Menu item for toggling the <i>LiveDirector</i> . . . . .	92
6.33	Virtual positions of the set cameras . . . . .	92
6.34	Diagram of the pan angle calculation . . . . .	93
6.35	Diagram of the tilt angle calculation . . . . .	94
6.36	Diagram of the camera's position calculation . . . . .	94
7.1	Topic classification analysis . . . . .	101
7.2	Overall classification of the developed solution . . . . .	102
7.3	Individual classification of the a) <i>StatisticsOverlay</i> and the b) <i>LiveCameraDirector</i> . . . . .	102



# List of Tables

2.1	PAGE Description of Agent Types . . . . .	8
2.2	Agent Capabilities (adapted from [26]) . . . . .	22
3.1	Raster vs Vector Graphics (adapted from [51]) . . . . .	29
7.1	Participant Characterization Questions . . . . .	98
7.2	Spearman’s Rank Order Correlation Results . . . . .	104





# Listings

3.1	Drawing a triangle in OpenGL . . . . .	30
3.2	Drawing a square in OpenGL . . . . .	35
3.3	Modifying the transform stack . . . . .	35
3.4	Rendering a full cube in OpenGL . . . . .	36
6.1	Configuration File Example . . . . .	67
6.2	Draw box method . . . . .	86
6.3	Draw dynamic panel box . . . . .	88
6.4	Render the statistical information text centered . . . . .	89



# Abbreviations

UDP	User Datagram Protocol
TCP	Transmission Control Protocol
IP	Internet Protocol
AI	Artificial Intelligence
CSG	Collective Sport Games
BDI	Belief-Desire-Intention
DAI	Distributed Artificial Intelligence
MAS	Multi-agent Systems
DPS	Distributed Problem Solving
KSE	Knowledge Sharing Effort
DARPA	Defence Advanced Research Projects Agency
KQML	Knowledge and Query Manipulation Language
KIF	Knowledge Interchange Format
FIPA	Foundation for intelligent Physical Agents
FIFA	International Federation of Association Football
API	Application Programming Interface
PBR	Polygon-Based-Rendering
MIT	Massachusetts Institute of Technology
CAD	Computer Aided Design
UI	User Interface



# Chapter 1

## Introduction

The present document shows the development of the dissertation made in the scope of the conclusion of the master's degree in Informatics and Computing Engineering. In this first chapter the context of the problem is described, in order to fit the environment in which all the work was developed, illustrating the main objectives proposed to be studied and also the entire structure of the dissertation, for easier analysis of the same.

### 1.1 Context and Motivation

A survey taken by the FIFA (International Federation of Association Football), the main entity in charge of creating and enforcing the rules of the sport around the world, in the summer of the turn of the century in the year 2000, revealed that 240 million people regularly play football around the world, along with almost five million referees, assistant referees and officials who are also directly involved in the game [15]. Furthermore, the statistics showed that over 20 million female footballers played the game. A number that has been rising and is now over 30 million worldwide [16]. All these athletes are distributed in a span of more than 1.5 million team and 300,000 clubs making it, by far, the most practiced sport in the world. Regarding the number of fans there is not an official statement made by a fully credible entity that discloses those numbers, but there are several websites which share around the same estimates for various sports, where soccer comes in first place with an estimated number of supporters in the proximity of 3.5 billion individuals [50]. If we sum up these numbers, we rapidly conclude that a large percentage of the world's population is somehow involved in the sport, making it the *King Sport*.

Other evolving trends in the world are the fields of Artificial Intelligence and Robotics and in order to stimulate advancements in these areas, several initiatives have been developed along several distinct domains that gather researchers from around the globe in order to solve complex problems and carry the technologies further. Following the main premise that soccer is one of the most engaging domains worldwide, RoboCup presented itself as a good candidate to prompt researchers to make progress on the fields. RoboCup is an international scientific initiative with the goal to advance the state of the art of intelligent robots. It has several leagues, each which a

distinct focus area on the field of Artificial Intelligence. One of its leagues is the RoboCup Soccer Simulation 3D League, which runs on its official simulation server, SimSpark - a generic physical multi-agent simulator system for agents in three-dimensional environments. Although providing a proper visualization of the agents' actions, it is unappealing and provides almost no information on what is happening on the pitch in statistical terms.

The environment of RoboCup Soccer is one of the most difficult for artificial intelligence researchers and presents several problems: an uncertain environment, multiple competitive agents, full physics, and the need for high-level cooperative behaviors. One of the greatest challenges in developing autonomous robotic agents is debugging and analyzing behaviors and algorithms. As such, there is a significant need for tools that assist researchers in understanding and developing their agents [46].

For the researchers - or coaches - it is important to have the possibility to analyze in real-time what is occurring on the pitch to understand how the teams are performing and to be able to see the most important movements and behaviors of their players, much like real-life soccer games.

## 1.2 Objectives and Approach

### 1.2.1 Thesis Question

The main question that originated the development of this thesis is

**How can we improve the visualization experience in a robotic simulated soccer match and provide valuable information about the match's events?**

More precisely, this thesis contributes to an open-source project which was developed by a team of event participating researchers and that is used as the 3D visualization tool for the simulated matches.

### 1.2.2 Approach

The approach for answering the thesis question is based on the creation of three modules that will make the visualization experience of these simulated matches more realistic and appealing to any spectator - researchers or bystanders. They are to be implemented in the official RoboCup 3D simulator, RoboViz, in order to possibly be used in the real-world competitions at some point in time. For that, we first created a fork of the GitHub repository of the project hosted by the *magmaOffenburg* team, with the intent to open a pull request with the new functionalities by the end of the development, testing and validation of this work.

The first and main component is an intelligent statistics module parser that is able to calculate several metrics usually applied to soccer - possession, shots on target, corners - and, to achieve that, information is gathered in real-time off of the simulator - ie. player and ball positions and speeds - and compiled into in-game events detection and valuable information that is then fed to both of the visualization modules. The second module is a statistics overlay and evaluates and displays the gathered information on screen at appropriate times given specific game events or

on demand through informative tables, graphs, grids and more. And lastly, an intelligent camera control module that allows for automatic selection and control of a set of cameras regarding agent's actions and the in-game events, granting a better vision of relevant occurrences in real-time, much like real human soccer matches live broadcasts.

The final result is an improved smart visualizer for the RoboCup 3D Soccer Simulation League making it more attractive and interesting to any spectator and allowing researchers to be able to produce a more well supported analysis of game and team tactics to then further improve their intelligent robots increasing their game performance and eventually outperforming their opponents as a multi-agent team.

## 1.3 Structure of the Dissertation

This remainder of this document is divided in four parts, as follows:

### 1. Part I: Literature Review

- (a) **Chapter 2** provides an extensive survey of two main areas transversal to the subject of this thesis, which are the fields of autonomous agents and multi-agent systems.
- (b) **Chapters 3 and 4** provide a survey about cinematography and graphical computation, respectively, which was crucial for the development of the visualization modules, exceptionally for the intelligent camera module.

### 2. Part II: Human Soccer and Robotic Soccer Review

- (a) **Chapter 5** contextualizes the setting upon which this dissertation's theme is developed by reviewing the RoboCup competition initiative thoroughly and also by exploring the evolution of soccer throughout the ages.

### 3. Part III: Modules Development

- (a) **Chapter 6** exposes the development of the modules set to be implemented in order of importance. Starting with the statistics gathering system followed by the visualization modules, namely the statistics screen overlay and the camera orchestration system.

### 4. Part IV: Work Validation

- (a) **Chapter 7** exposes the results of a questionnaire sent to individuals of multiple fields and general population overall to evaluate the developed tool, as well as some data analysis on them.

### 5. Part V: Closing Remarks

- (a) **Chapter 8** summarizes the contributions of this work to the field and discusses some future applications and possibles developments on top of the project.





## Chapter 2

# Autonomous Agents and Multi Agent Systems

Multi Agent Systems form a sub-area of Distributed Artificial Intelligence with main focus on the interaction between multiple intelligent computing elements denominated by *agents* which communicate in order to solve problems that are beyond the individual capabilities or knowledge of each individual. Despite there being no categorical definition of MAS, a largely consensual one can be that "a loosely coupled network of problem-solving entities (agents) that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity (agent)" [47]. It has been studied as a field of its own since the 1980's, gaining widespread recognition through the 90's and reaching enormous growth up until nowadays with the incredible evolution of computational units and processing power leading to ground-breaking and innovative technologies in the most diverse areas of society.

According to K. Sycara [48], multi agent systems must have present four underlying characteristics:

- Each agent has incomplete information or capabilities for solving the problem and, thus, has a limited viewpoint.
- There is no system's global control.
- Data is decentralized.
- Computational is asynchronous and requires coordination.

Building upon the previous definition, it becomes clear that multi agent systems require a great deal of communication, coordination and negotiation among its parts (the agents) in order to achieve maximum efficiency when operating towards the system's goals. Each agent can be more effective in the context of others because it can focus on tasks it excels at, delegate other tasks and negotiate actions in order to achieve its goals and, ultimately, the system's goals.

In this chapter I will explain more in depth the concepts of *agent* and *MAS* to the level that is required in order to understand the underlying system upon which this dissertation project will

be developed. We will analyse the most agreed upon agent definitions, main attributes and most current architectures and, in the MAS domain, we will explore real-world applications (like the RoboCup project) and get a better overview of inter-agent communication and coordination protocols within such a system.

## 2.1 Intelligent Agents

### 2.1.1 Agent Definition

In a very broad manner, as stated by Russel and Norvig, "an agent is anything that can be considered able to perceive its environment through sensors and act on this environment through actuators" [41]. In a more software oriented environment, an agent is understood as a single computational system based on software that, in its bare minimum, has at least the following properties according to a well accepted definition by Woolridge and Jennings [56]:

**Autonomy** - It operates without any human or other agent direct intervention and has control of its own internal state and taken actions.

**Reactivity** - The agents are ware of their environment and are capable of reacting to changes on it.

**Pro-Activity** - Not only they react to environment changes, they are capable of behaving accordingly to achieve their desired goals.

**Social Abilities** - The agents are able to interact with other agents and exchange communications through well defined communication languages.

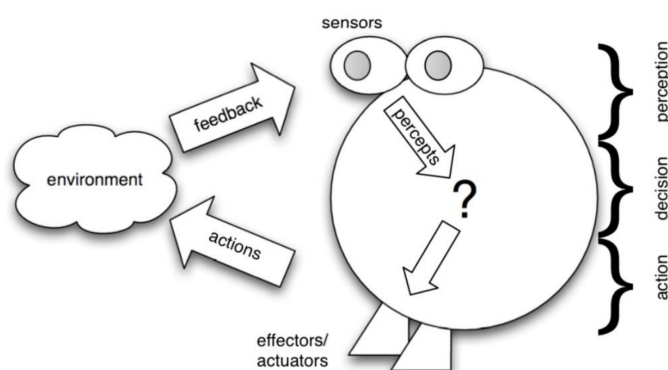


Figure 2.1: An agent in its environment (adapted from [41]). The agent takes sensory input from the environment and produces, as output, actions that affect it. [55]

Autonomy is one of the most consensual properties in the field's community but, for some, it would never be fully achieved. It is clear that the agent has to be created and put into operation by another party (human or agent). The assumption that the agent action will not have an end is

also not entirely valid. The agent will have a limited lifetime and a final action. On the other hand, although autonomy is essential to the agent, usually human-agent interaction is desirable or even essential. It is usual to build agents that behave autonomously but are also able to take orders or instructions from humans [29].

Reactive and proactive behaviours should balance one another since neither a purely reactive agent nor an extremely proactive one would thrive and out-perform one with such capabilities of adaptation and action taking. A suitable balance of these properties is crucial for the performance of an individual agent within a given environment, which should be heavily taken into account when tuning this behavioural responses - a dynamic environment would be best populated by more reactive agents and a more static environment could be more focused on proactive responses.

The social abilities of an agent would, in many ways, relate directly to those of we humans in a sense that, for its long-term success, it is essential that it is able to exchange high-level messages and carry out processes of social interaction with others in its environment [29]. These processes may include coordination, cooperation, competition and/or negotiation and play a substantial role in a system where its agents often may have different goals and/or environment perceptions.

A side from their behaviour, one must talk broadly how they work on the inside. In general, the agent's architecture - some sort of computing device - makes the perceptions from the sensors available to its program - the function that implements the agent's mapping from perceptions to actions. The relationship among agents, architectures and programs, according to Russel and Norvig, can be summed up as [41]:

$$agent = architecture + program$$

However, before starting to design an agent's program, we must first have a solid idea of its possible perceptions and desired actions, what goals or performance measure the agent is to achieve, and what sort of environment it will operate in. The authors call this the PAGE description (Percepts, Actions, Goals, Environment). Table 2.1 describes examples of a few real-world agent types following the structure derived by Russel and Norvig [41].

### 2.1.2 Agent Programs

An agent program can be simply defined mathematically as a function  $f$  which maps every possible perceptual input sequence of the agent at any instant to a possible action it can perform or to a coefficient, function or constant that can affect other prospective actions.

$$f : P^* \rightarrow A \tag{2.1}$$

In their work, Russel and Norvig defined a *SKELETON-PROGRAM* (Algorithm 1) which all agents built throughout the book implemented. It was over-simplified, but it served to illustrate the main idea of the flow that should be taken into consideration when building any kind of agent and writing its program.

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis systems	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Autonomous driving systems	Color, distances, global position, car sensors	Control engine speed, steer, calculate decisions	Move the vehicle safely from A to B	Public roads, pedestrians, other vehicles
Inventory control system	Sales forecast, existing stock	Stockpile, liquidate, replenish	Minimize storage cost	Inventory and sales databases, user

Table 2.1: PAGE Description of Agent Types

**Algorithm 1:** SKELETON-AGENT (adapted from [41])**Input:** *percept***Result:** *action***Data:** *memory*, the agent's memory of the world

```

1 memory ← UPDATE – MEMORY(memory, percept);
2 action ← CHOOSE – BEST – ACTION(memory);
3 memory ← UPDATE – MEMORY(memory, action);
4 return action;

```

In the algorithm, the agent receives only one percept, although it has been defined that the agent program receives percept *sequences*. It is up to the agent to build up the percept sequence in memory. In some environments it is possible to be successful without storing the percept sequence, and in complex domains, it is actually infeasible to store it in its entirety.

As a software program, an agent program, which maps percepts to actions, in the process of decision, according to Khosla, Sethi and Damiani [38], commonly exhibits some characteristics, as follows:

**Autonomy** - An agent should be able to exercise a degree of autonomy in its operations. It should be able to take initiative and exercise a non-trivial degree of control over its own actions.

**Collaboration** - An agent should have the ability to collaborate and exchange information with other agents in the environment to assist other agents in improving their quality of decision making as well as its own.

**Flexibility and Versatility** - An agent should be able to dynamically choose which actions to invoke, and in what sequence, in response to the state of its methods from which it can

formulate its actions and action sequences. This facility provides versatility as well as more flexibility to respond to new situations and new contexts.

**Temporal History** - An agent should be able to keep a record of its beliefs and internal state and other information about the state of its continuously changing environment. The record of its internal state helps it to achieve its goals as well as revise its previous decisions in light of new data from the environment.

**Adaptation and Learning** - An agent should have the capability to adapt to new situations in its environment. This includes the capability to learn from new situations and not repeat its mistakes.

**Knowledge Representation** - In order to support its actions and goals with an agent, it should have the capabilities and constructs to properly model structural and relational information of the problem domain and its environment.

**Communication** - An agent should be able to engage in complex communication with other agents, including human agents, in order to obtain information or request for their help in accomplishing its goals.

**Distributed and Continuous Operation** - An agent should be capable of distributed and continuous operation (even without human intervention) in one machine as well as across different machines for accomplishing its goals.

### 2.1.3 Agent Autonomy

As talked before in this section, **autonomy** is one of the most consensual agent attributes along the literature. As Russel and Norvig put it, if an agent's actions are based completely on built-in knowledge (pre-programmed facts that the system designer included in its program), then one would say that the agent lacks autonomy.

"An agent is autonomous to the extent that its behaviour is determined by its own experience" [41]. However, to require an agent to be capable of full autonomy from the very beginning of its deployment, is an unquestionably hard demand. So, just as evolution provides animals with enough built-in reflexes so that they can survive long enough to learn for themselves, it would be reasonable to provide an artificial intelligent agent with some initial knowledge as well as an ability to learn [41].

Following on the same thought, Wooldridge and Jennings defend in their work that agent autonomy also evolves with the increase in pro-activity behaviour of itself [56], since it will behave differently according to its ever updating perception of the environment.

Hexmoore, Castelfranchi and Falcone, in their work entitled "A Prospectus on Agent Autonomy" [21], studied two types of interaction on the subject of agent autonomy. The first being the *human-machine* interaction. In this type, autonomy concerns are predominantly for the agent to acquire and to adapt to human preferences and guidance. The main reference point here is always

the human and the definition of a fully autonomous agent is when it has access to the complete set of choices and preferences of the distinguished entity that can judge or change the agent's autonomy - *adjustable autonomy* - so that, this way, the agent could behave as its authority would in each individual situation.

The second type is *agent-to-agent* interactions, which that advantage of *relative autonomy*. In these interactions an agent's autonomy can be relative to another agent or environmental factor. There is no user but any other agent may be the reference point [21]. This way, an agent can be the entity of which choices and preferences would influence the actions and behaviours of other agents that agreed to be controlled over them. In these scenarios, the autonomy of the agents that agreed to be the subjected to control, is lower than the agent that is controlling. Control influences autonomy, however, the inverse does not hold.

### 2.1.4 Agent Environment

An agent is normally, by definition, included on a surrounding environment, either a real physical one or a simulated space. The environment of an agent has direct influence over most aspects of its existence - actions, behaviours - and, as such, it impacts directly the development of its program as the complexity of the action choice process can be affected by several characteristics of its environment. The main recognized properties of an environment as those suggested by Russel and Norvig in 1995 [41]:

**Accessible vs Inaccessible** - If an agent's sensors give it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

**Deterministic vs Non-deterministic** - If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic.

**Episodic vs Non-episodic** - In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

**Static vs Dynamic** - If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is semidynamic.

**Discrete vs Continuous** - If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete.

In general, the most complex environment is that which is comprised of the *inaccessible*, *non-episodic*, *dynamic* and *continuous* properties. The *non-deterministic* property is normally assumed on most real situations due to the fact that their complexity causes the possibility of it being actually treated as *deterministic* to be very slim.

However, despite the type of environment and its combination of properties, the system that would simulate its existence and organization would most definitely follow always the same generic algorithm that would be responsible for delivering each agent its percepts, retrieving its action and updating the state accordingly. As before, Russel and Norvig have illustrated this procedure for better understanding (Algorithm 2).

---

**Algorithm 2:** RUN-ENVIRONMENT (adapted from [41])

---

**Input:** *state*, the initial state of the environment  
**Input:** UPDATE-FN, function to modify the environment  
**Input:** *agents*, a set of agents  
**Input:** *termination*, a predicate to test when we are done

```

1 repeat
2   foreach agent in agents do
3     PERCEPT[agent] ← GET – PERCEPT(agent, state)
   end
4   foreach agent in agents do
5     ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
   end
6   state ← UPDATE – FN(actions, agents, state)
until termination(state);

```

---

### 2.1.5 Agent Architecture

A software architecture could be defined as a component configuration that is part of a system and also the connection that coordinates the activities between the components [18]. Figure 2.2 illustrates how an agent's base-architecture is constructed where it, at its simplest form, follows a linear and unidirectional path from input (percepts) to output (action).

A good architecture should reflect the concepts of **Simplicity**, **Features**, **Expansivity** and **Portability** as defined by Mowbray [49] and, although the basis for a strong architecture would remain close to the aforementioned, for a specific agent with certain capabilities and distinct goals, the architectures would take different shapes.

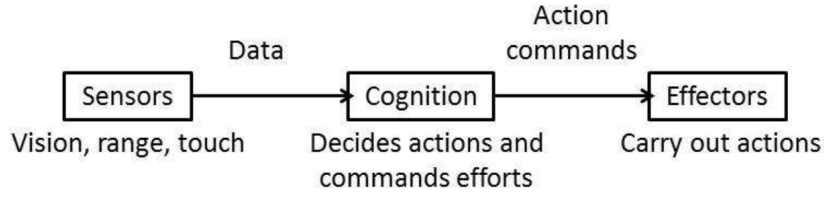


Figure 2.2: Basic Architecture of an Autonomous Agent (adapted from [32])

In their work, Russel and Norvig [41] consider just four distinct types of agent programs - **Simple Reflex agents**, **Model-based Reflex agents**, **Goal-based agents**, **Utility-based agents** - where each has its own recognizable advantages for certain objectives.

### 2.1.5.1 Simple Reflex Agents

This types of programs are purely reactive, where the agent has an initial pre-populated set of percept-to-action mappings that, given an environment perception, will execute the pre-established mapped action. These type of agents are among the simplest to program (Algorithm 3) as they work by finding a rule whose condition matches the current situation and then simply executing it. However, they operate best in static and most-deterministic environments making them suitable only for well specified settings.

---

#### Algorithm 3: SIMPLE-REFLEX-AGENT (adapted from [41])

---

**Input:** *percept*

**Result:** *action*

**Data:** *rules*, a set of condition-action rules

- 1 *state*  $\leftarrow$  INTERPRET – INPUT(*percept*);
  - 2 *rule*  $\leftarrow$  RULE – MATCH(*state*, *rules*);
  - 3 *action*  $\leftarrow$  RULE – ACTION[*rule*];
  - 4 **return** *action*;
- 

### 2.1.5.2 Model-based Reflex Agents

A model-based reflex agents acts, in much similar way, the same as a simple-reflex agent. However, it is capable of taking a step further when the environment it is inserted into is subject to evolution and time-based changes. This type of architecture considers some sort of internal state has to be maintained - percept history and internal state memory - by the agent in order to be able to choose the correct action to take. It is still a reactive agent but also takes into account how its own actions affect the state of the world, as illustrated by algorithm 4.



**Algorithm 4:** REFLEX-AGENT-WITH-STATE (adapted from [41])

---

**Input:** *percept*  
**Result:** *action*  
**Data:** *rules*, a set of condition-action rules  
**Data:** *state*, a description of the current world state

```

1 state  $\leftarrow$  UPDATE – STATE(state, percept);
2 rule  $\leftarrow$  RULE – MATCH(state, rules);
3 action  $\leftarrow$  RULE – ACTION[rule];
4 state  $\leftarrow$  UPDATE – STATE(state, action);
5 return action;

```

---

**2.1.5.3 Goal-based Agents**

Being aware of the current state of the environment or of its immediate last state, may not be sufficient to make a decision on the next action to take in order to be able to achieve a given goal. The agent needs information about its goal and on how its actions can influence the world around it (same as the mode-based reflex agent). The process can be of easy calculation or, depending on the situation at hands, may require the agent to be able to consider long sequences of actions to determine the best way to achieve its goal. As such, this type of agent has more complex decision-taking processes since it has to evaluate both what would happen by taking a given action and if the end result of that choice would eventually lead it towards it's goal ("make it happy") 2.3.

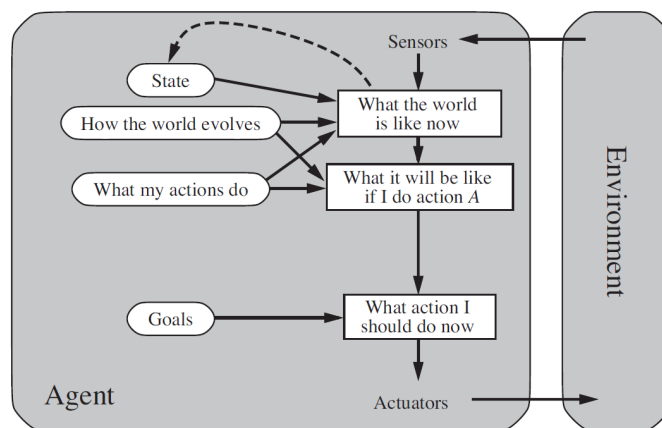


Figure 2.3: A goal-based agent architecture where it has explicit goals (adapted from [41]).

**2.1.5.4 Utility-based Agents**

Following on the previous architecture, goal-based, one should note that simply taking into consideration the end result (complete the goal) and acting accordingly is not adequate to achieve a high-quality behaviour. By high-quality behaviour we can assume one of different metrics or a combination of some, such as *time*, *costs*, *safety*, *reliability* or others. The previous approach only

evaluates if the end result of taking the chosen set of actions leads the agent to its goal, a boolean result, - yes or no - whereas an utility-based agent does also take into account some performance measure that would be able to quantify exactly "how happy" the end result would make the agent or, in a more scientifically accepted terminology, which final state would hold a higher **utility** for the agent 2.4.

Utility is, therefore a function that maps a given state onto a real quantifiable number that would represent would "happy" the agent would be in such a state. The agent could then choose what set of actions it should take in order to achieve either the highest utility final state or a set of intermediate states that would ultimately lead it to a more desirable end result, the decision depends on the approach as well as the problem at hands in each individual situation.

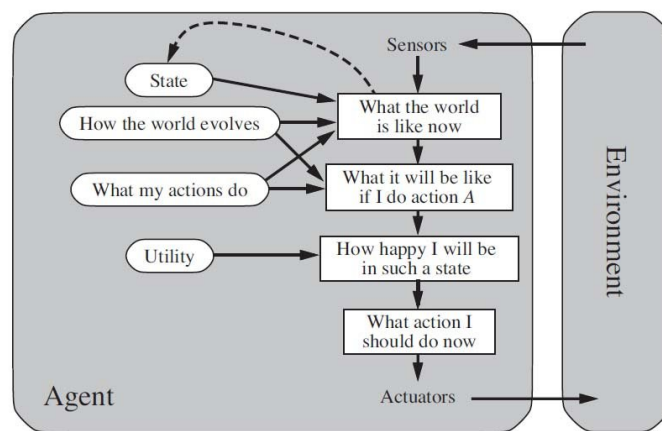


Figure 2.4: An utility-based agent architecture where it maps a given state to an actionable measurement (adapted from [41]).

On a more generic approach to agent architecture, there are three agreed upon base architectural groups for building agents and multi-agent systems - Deliberative, Reactive and Hybrid - that each developed agent architecture can be, in its essence, mapped to and that will be discussed further according to their evolution throughout history.

### 2.1.5.5 Deliberative Architecture

The deliberative architecture, also known as logic-based or symbolic-based architecture, is one of the earliest agent architectures that was the traditional approach to design all agents designed within AI during the early stages of the domain.

Agents implementing this architecture use explicit reasoning in order to determine how to act. Reasoning can be either **theoretical** (directed towards beliefs) or **practical** (directed towards actions). In order to be able to do so, the architecture must be one that contains an explicitly represented, symbolic model of the world and that is able to make decisions on what action to take via symbolic reasoning. The syntactical manipulation of the symbolic representation is the process of logical deduction or theorem proving. As an instance of theorem proving, the agent

specifications outlines how the agent behaves, how the goals are generated and what action the agent can take to satisfy these goals [36].

### 2.1.5.6 Reactive Architecture

As some researchers felt that there were many unsolved problem associated with symbolic AI, they were led to develop different systems and ended up developing reactive architectures. Reactive agent architectures consists on the direct mapping of a environment situation (sequence of percepts) to an action. No symbolic world model and reasoning are used as the agent responses to environment evolution are purely based on its direct perceptual input.

The best example of such an architecture is Brook's subsumption architecture. The key idea of subsumption architecture is that intelligent behaviour can be generated without explicit representations and abstract reasoning with symbolic AI technique [7]. A subsumption architecture is a hierarchy of task-accomplishing behaviours, where the lowest layer has the utmost priority, where each of them has simple input-to-action mappings and complex behaviour can be achieve through their combination (represented in 2.5).

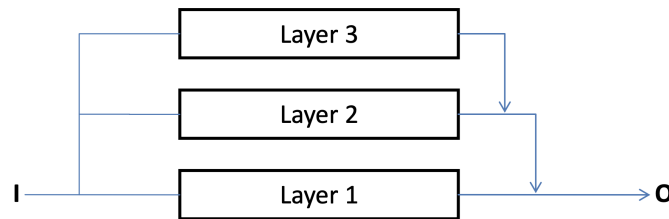


Figure 2.5: Actions selection in a layered subsumption architecture (adapted from [36])

### 2.1.5.7 Hybrid Architecture

Often with almost everything in existence, neither extremity of an implementation spectrum achieves the best performances and, as such, a balance between options is usually the optimal approach. Has some may argue, the same applies to building agents which could probably perform best by implementing subsystems of the previously discussed architectures.

The main problem with this approach is to determine how to implement the main system that would control the interactions between the architecture's layers. Two generic proposals are [55]:

**Horizontal layering** - In horizontally layered architectures (see part (a) of Figure 2.6), the software layers are each directly connected to the sensory input and action output. In effect, each layer itself acts like an agent, producing suggestions as to what action to perform.

**Vertical layering** - In vertically layered architectures (see parts (b) and (c) of Figure 2.6), sensory input and action output are each dealt with by at most one layer.

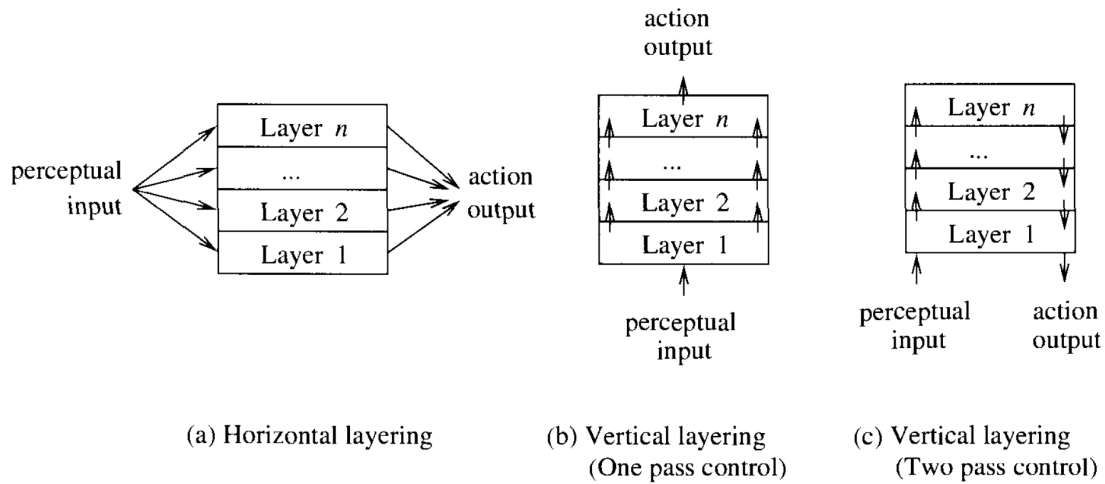


Figure 2.6: Information and control flows in three types of layered agent architecture (adapted from [55])

#### 2.1.5.8 Belief-Desire-Intention (BDI) Architecture

The BDI architecture is based on practical reasoning (introduced in 2.1.5.5) which is reasoning directed towards actions, figuring out how to act. Being a reasoning-based architecture, it can be classified as a subset of deliberative architectures.

Human practical reasoning involves two activities namely deliberation and means-end reasoning. Deliberation decides what state of affairs needs to be achieved while means-end reasoning decides how to achieve these states of affairs [36]. The agent's architecture consists of three logic components internally connected (see figure 2.7):

**Beliefs** - Reflect the knowledge of the agent about the world, including itself and other agents.

**Desires** - Represent the agent's motivational state, its objectives and goals.

**Intentions** - The agent commitment towards its desires and beliefs, what it has chosen to do. These are a key part of the practical reasoning implementation.

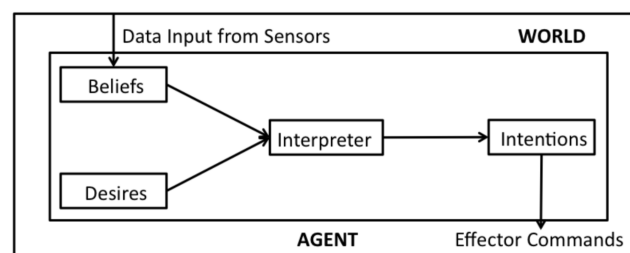


Figure 2.7: A BDI Agent Architecture (from [12])

## 2.2 MAS - Multi-Agent Systems

Multi-agent systems are a sub-area of the Distributed Artificial Intelligence (DAI) domain where computational systems are created based on the single entities - the agents - giving place to semi-autonomous agent societies that interact through a common environment where they actuate on, altering its internal state. These systems are normally distributed, therefore coordination became a major area of interest for research. In the end, there are two major groups coordination methodologies: methodologies for competitive agent domains and for cooperative agent domains which can be used in conjunction or disjunction within the same system in order to obtain the desired results.

A MAS is constituted by multiple agents with different perceptions and action capabilities in a specific environment. Each agent is capable to influence a distinct part of the world [30] so it is crucial for the individuals to be able to communicate efficiently with one another to be able to exchange information and proceed towards achieving their individual or community goals and objectives and, as such, the architectural implementation of each of the existing agents can differ in many ways from each other. An overview of a basic implementation of a multi-agent system can be seen in Figure 2.8.

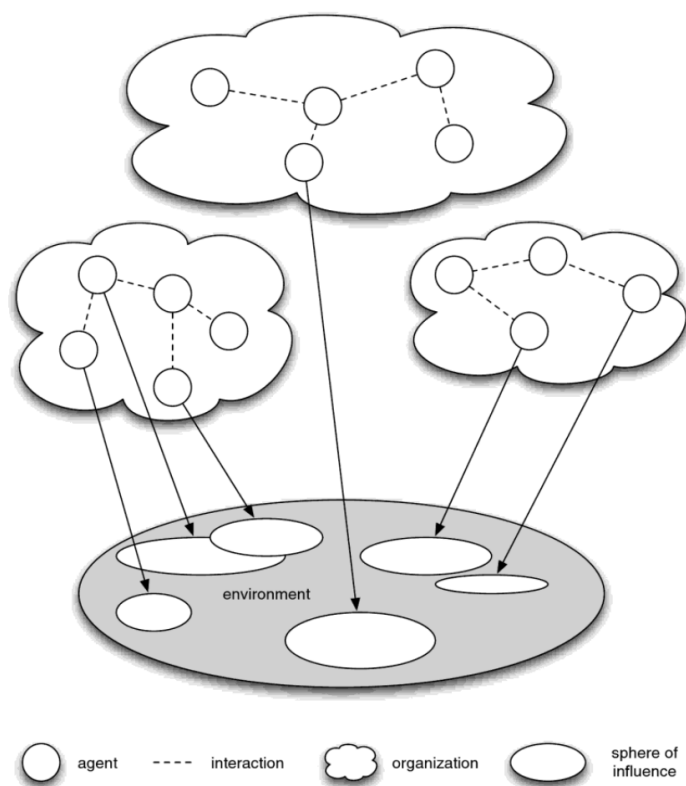


Figure 2.8: Basic Architecture of a Multi-agent system (from [55])

### 2.2.1 Distributed Problem Solving vs Multi-agent systems

Both Distributed Problem Solving (DPS) and Multi-agent Systems (MAS) are systems composed of multiple interacting agents, causing newcomers on to the field to often mistakenly refer to both using the "multi-agent systems" nomenclature, as well as to many terms inside the field. However, they are distinct branches of the parenting DAI field that have emerged at different points in time and have then unknowingly started to overlap definition.

The main goal of both remains under the umbrella of their parent, DAI, which is essentially focused on problem resolution through individual processing units. Bond and Gasser described both DPS and MAS as the primary arenas within DAI due to the aforementioned falsely overlap for some and describes their main differences [6] (see figure 2.9):

**Distributed Problem Solving** - The work of solving a particular problem can be divided among a number of modules, or "nodes", that cooperate at the level of dividing and sharing knowledge about the problem and the developing solution.

**Multi-agent Systems** - With MAS, the research is concerned with coordinating intelligent behaviour among a collection of autonomous intelligent *agents*. The agents in this type of system may be working toward a single global goal, or toward separate individual goals. They, like DPS, must share knowledge about the problems and solutions, but must also reason about the processes of coordination among themselves.

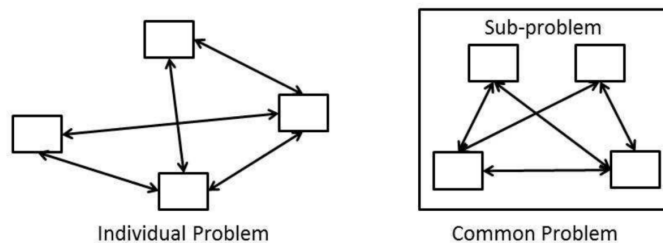


Figure 2.9: (a) Multi Agent System (b) Distributed Problem Solving (from [12])

### 2.2.2 Multi-agent Systems Motivation

The wide adoption of multi-agent systems in multiple application domains is due to the many beneficial advantages its implementation offers, mainly in the dimension of performance like [48]:

**Computational Efficiency** - Concurrency and asynchronous capabilities allow for faster and more efficient results.

**Reliability** - It allows for graceful recovery of component failures with the use of agents with redundant capabilities and through appropriate coordination.

**Extensibility** - The number and capabilities of agents working on a given problem can be easily altered at any point in time.

**Robustness** - The system's ability to tolerate uncertainty is higher because suitable information is exchanged among agents.

**Maintainability** - The inherent modularity of a multi-agent system makes it easy to maintain.

**Responsiveness** - Because modularity can handle anomalies locally, they would not be propagated to the whole system.

**Flexibility** - The existence of agents with different abilities can adaptively organize to solve the current problem.

**Reusability** - Due to modularity, functionally specific agents can be reused in different agent teams to solve different problems.

In the intrinsic nature of a distributed systems, even more reasons can be listed on why multi-agent systems are highly in use for problem solving [12]:

1. The problem dimension is too high to be solved by an single agent;
2. Allow the interconnection and interoperability of multiple legacy systems;
3. Provide a natural solution for geographical and/or distributed problems;
4. Confer simplicity in the conceptual project;
5. Allow cooperative man-machine interface in which both act as agents in the system;
6. Supply problem resolutions in which the experts and the knowledge (to solve the problem) are distributed;

### 2.2.3 Agent Communication

As discussed in several occasions along this chapter, communication between computational entities is an area of great focus within the DAI field and it has always been considered one of the most important problems in computer science. However, in the area of Multi-Agent Systems, communication is treated at a much higher level than in other areas of computer science. Communication has two main purposes: sharing knowledge, information, beliefs or plans with other agents; and coordination of activities between agents [40].

### 2.2.3.1 Software Architectures for Communication

Two very distinct approaches have been implemented through the years: *direct communication* and *assisted coordination*.

**Direct Communication** - In direct communication architectures, agents handle their own coordination. There are two main approaches for implementation: *contract-net* - where agents in need of service distribute requests for proposals to other agents who then bid on the requests for then the originator to decide on who to award the contract to - and *specification sharing* - where agents supply other agents with information about their capabilities and needs, and these agents can then use this information to coordinate their activities. A disadvantage of direct communication is its cost when the number of cooperating agents increases. With a large number of programs the cost of broadcasting the bids or specifications and consequential processing of those messages is prohibitive. Another disadvantage is implementation complexity. In the direct communication schemes, each agent is responsible for negotiating with other agents and must contain all of the code necessary to support this negotiation [19].

**Assisted Coordination** - A popular alternative to direct communication is to organize agents into what is often called a *federated system* (see fig. 2.10). In this approach, agents do not communicate directly but they communicate only with system programs called facilitators (or mediators). In this system, agents use agent communication language to document their needs and abilities for their local facilitators. In addition to this they also send application-level information and request to their facilitators and accept application-level information and requests in return. Facilitators then use the documentation provided by these agents to transform these application-level messages and route them to the appropriate place [19].

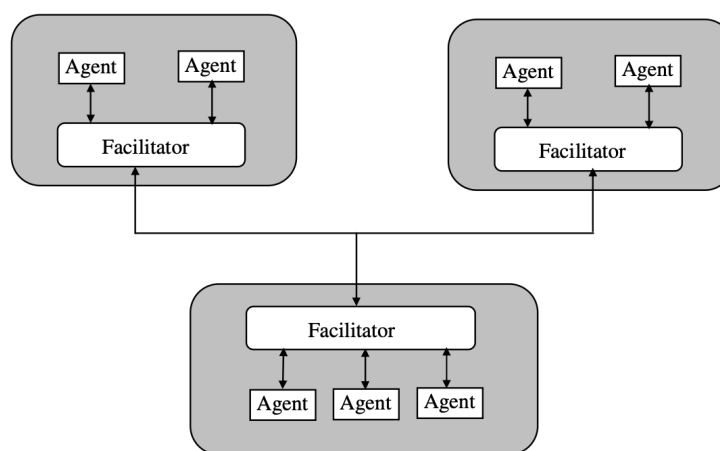


Figure 2.10: A federated system architecture (from [19])



### 2.2.3.2 Dimensions of Meaning

Agents communicate in order to understand and be understood, so it is important to consider the different dimensions of meaning that are associated with communication, them being [45]:

**Syntax** - How the communication's symbols are structured, it contains the rule set related to word combinations.

**Semantics** - What the symbols denote, what is the meaning of the given words and statements.

**Pragmatics** - How the symbols are interpreted.

In the end, the meaning of the information is the combination of both the semantics and pragmatics aspects and it follows some common characteristics [45].

**Descriptive vs Prescriptive** - Some messages describe phenomena, while others prescribe behavior. Descriptions are important for human comprehension, but are difficult for agents to mimic so most agent communication languages are designed for the exchange of information about activities and behavior.

**Personal vs Conventional Meaning** - An agent might have its own meaning for a message, but this might differ from the meaning conventionally accepted by the other agents. In MAS, the system should opt for conventional meanings, in particular to deal with the entrance of new agents.

**Subjective vs Objective Meaning** - Similar to conventional meaning, where meaning is determined external to an agent, a message often has an explicit effect on the environment, which can be perceived objectively. The effect might be different than that understood internally, subjectively.

**Speaker's vs Society's Perspective** - Independent of the conventional or objective meaning of a message, the message can be expressed according to the viewpoint of the speaker or hearer or other observers.

**Semantics vs Pragmatics** - The pragmatics of a communication are concerned with how the communicators use the communication. This includes considerations of the mental states of the communicators and the environment in which they exist, considerations that are external to the syntax and semantics of the communication.

**Contextuality** - Messages cannot be understood in isolation, but must be interpreted in terms of the mental states of the agents, the present state of the environment, and the environment's history. Interpretations are directly affected by previous messages and actions of the agents.

**Coverage** - Smaller languages are more manageable, but they must be large enough so that an agent can convey the meanings it intends.

**Identity** - When a communication occurs among agents, its meaning is dependent on the identities and roles of the agents involved, and on how the involved agents are specified. A message might be sent to a particular agent, or to just any agent satisfying a specified criterion.

**Cardinality** - A message sent privately to one agent would be understood differently than the same message broadcast publicly.

### 2.2.3.3 Message Types

Agents of different capabilities should be able to communicate therefore, according to Huhns and Stephens, communication must be defined at several levels, with communication at the lowest level used for communication with the least capable agent. In order to be of interest to each other, the agents must be able to participate in a dialogue. Their role in this dialogue may be either active, passive, or both, allowing them to function as a master, slave, or peer, respectively [26].

In its root form, there are two types of messages, assertions and queries, and according to the capabilities of each agent there can be considered to exist four distinct groups where each would fit (see table 2.2).

	Basic Agent	Passive Agent	Active Agent	Peer Agent
Receives assertions	X	X	X	X
Receives queries		X		X
Sends assertions		X	X	X
Sends queries			X	X

Table 2.2: Agent Capabilities (adapted from [26])

### 2.2.3.4 Protocols and Communication Levels

Usually the protocol's definition involves various levels [26]. The inferior level is related to the interconnection agent level while the intermediate level defines the transmitted information (syntax). Finally, the superior level defines the information specification (semantic). Generally, a protocol follows a specific data structure constituted by a sender, one or more receivers, a used language with its respective coding and decoding functions and also a set of actions that a receiver must execute [26].

### 2.2.3.5 Communication Languages

According to T. Finn et al., a suitable agent communication language should follow seven requirements where its overall value could be measured to the extent that it meets those requirements. They are as follows [17]:

**Form** - A good agent communication language should be declarative, syntactically simple, and readable by people. Finally, its syntax should be extensible.

**Content** - The language should commit to a well defined set of communicative acts (primitives). The choice of the core set of primitives also relates to the decision of whether to commit to a specific content language since committing allows for a more restricted set of communicative acts because it is then possible to carry more information at the content language level.

**Semantics** - Although the semantic description of communication languages and their primitives is often limited to natural language descriptions, a more formal description is necessary if the communication language is intended for interaction among a diverse range of applications.

**Implementation** - The implementation should be efficient, both in speed and in bandwidth utilization. It should provide a good fit with existing software technology and the interface should be easy to use.

**Networking** - An agent communication language should fit well with modern networking technology. The language should support all of the basic connections - point-to-point, multicast and broadcast - in both the form of synchronous and asynchronous connections.

**Environment** - The environment in which intelligent agents will be required to work will be highly distributed, heterogeneous, and extremely dynamic. Therefore, a communication language must provide tools for coping with heterogeneity and dynamism. It must support interoperability with other languages and protocols.

**Reliability** - A communication language must support reliable and secure communication among agents.

In the late 80's the Knowledge Sharing Effort (KSE) was founded in the USA financed by DARPA (Defence Advanced Research Projects Agency) which had the goal to develop protocols for the exchange and information representation between autonomous information systems [12]. The KSE developed two products:

**KQML (Knowledge and Query Manipulation Language)** - KQML was conceived as both a message format and a message-handling protocol to support run-time knowledge sharing among agents. It can be thought of as consisting of three layers: the content layer,

the message layer, and the communication layer. The content layer bears the actual content of the message, in the programs own representation language. The message layer forms the core of the KQML language, and determines the kinds of interactions one can have with a KQML-speaking agent and also has the ability to identify the protocol to be used to deliver the message and to supply a speech act or performative which the sender attaches to the content - assertion, query, command, etc [17].

**KIF (Knowledge Interchange Format)** - KIF is the solution suggested by the KSE for the syntactic aspects of representations for knowledge sharing. The language is intended as a powerful vehicle to express knowledge and meta knowledge. It is a prefix version of first order predicate calculus with extensions to support non-monotonic reasoning and definitions. The language description includes both a specification for its syntax and one for its semantics [17].

In addition to KQML and KIF, there are several languages defined in the context of communication in Multi-Agent Systems. Among the most used, the *FIPA ACL (Agent Communication Language)* stands out. In 1995, *FIPA - Foundation for Intelligent Physical Agents* started developing standards for Multi-Agent Systems. The resulting *ACL* is similar to *KQML*, being primarily an external communication language and does not require the use of any specific language for the content.

#### 2.2.3.6 The 'Social Agency' Communication Model

In conventional multi-agent systems based on mental attitudes like belief, intention, and commitment, agent communication suffers from the lack of a concise and universally accepted formal semantics. As a result, agent communication is confined in the realm of restricted environments and heterogeneous agents do not interact [35]. A proposed solution for this drawback was presented by M. Singh provides a foundation for the design of multi-agent systems where he presents that the success of such a system relies on the underlying ACL and if whether or not it supports the interaction among agents in a social setting. The elements said to contribute to the meaning of communication between agents can be depicted on figure 2.11.

**Perspective** - It can be private, meaning that it stays within the individual agent perspective (normally senders or receivers), or public (agent societies) and more according to MAS. The message sent contains knowledge and attitudes about senders only, or some shared knowledge of the multi-agent system.

**Type of Meaning** - Meaning is either individual or conventional, where a more personal perspective is related to the interpretation of the communication act between the sender and the receiver and a conventional is based on usage conventions.

**Basis** - Semantics are related to the language it self and its internal meaning as pragmatics include external considerations to the proper language as well as to the environment.

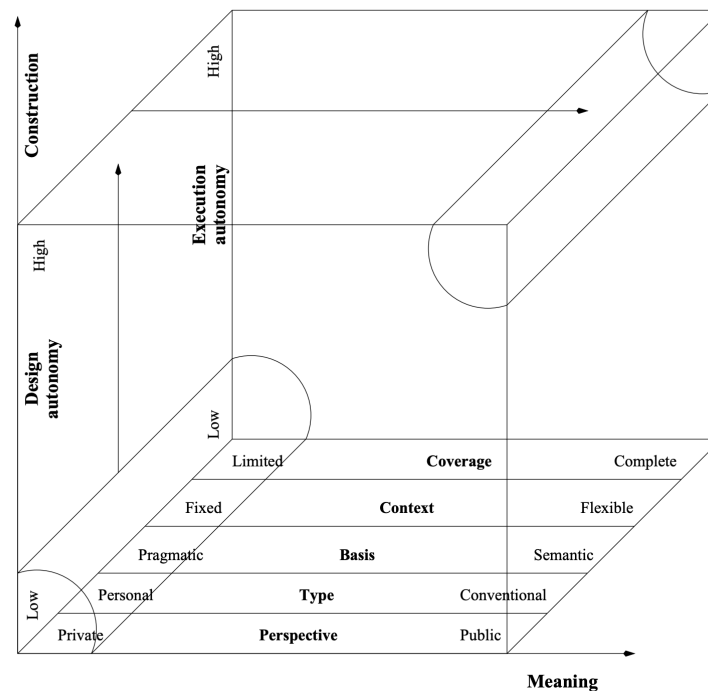


Figure 2.11: The design space of agent communication languages. The region in the left represents existing ACLs, which follows a mental agency model. The region in the upper right represents the desired goals, which dictate a social agency model (adapted from [35]).

**Context** - To understand a language it is also necessary to analyse the context it is inserted on. Therefore it should be flexible to allow a more meaningful agent communication between outer entities.

**Coverage** - When information is exchanged, the meaning of the message is characterized by communicative acts. The coverage of these acts should be wide in order to improve interactions within a multi-agent system.

### 2.2.3.7 Multi-agent Systems Coordination

In an environment populated by many distinct, heterogeneous and individual agents, their coordination is a central concern. The lack of or a poor coordination between the parts will eventually lead to a misuse of each individual's efforts and resources in the quest to achieve the individual or global goal. Essentially, co-ordination is a property of an intelligent agency that ensures that individual agents act in a coherent manner [14]. Among several reasons that depict the importance of coordination in multi-agent systems, some are the prevention of anarchy and chaos of the system, meeting global constraints, sharing of results and efficiency, for example. Note that coordination does not imply cooperation: an effective competitor will coordinate decisions to maximize its advantage against an opponent.

Regarding the implementation of such coordinating systems, Nwana, Lee and Jennings classified coordination techniques in four broad categories:

**Organizational Structuring** - This is the simplest co-ordination scenario which exploits the *a priori* organisational structure. This is because the organization defines implicitly the agent's responsibilities, capabilities, connectivity and control flow. It provides a framework for activity and interaction through the definition of roles, communication paths and authority relationships [43].

**Contracting** - In this approach, which assumes a decentralized bidding market structure, agents can assume two roles: a **manager** - who breaks a problem into sub-problems and searches for contractors to do them, as well as to monitor the problem's overall solution - or a **contractor** [43].

**Multi-agent planning** - In order to avoid inconsistent or conflicting actions and interactions, agents build a multi-agent plan that details all the future actions and interactions required to achieve their goals, and interleave execution with more planning and re-planning [43]. It can either be implemented in a decentralized manner or distributed.

**Negotiation** - Is an underlying part of almost all coordination techniques and can be defined as a communication process between a group in order to reach an agreement on some matter.

## 2.3 Conclusions

The DAI field is in constant evolution and, with it, so is the robotics fields. A key component in all of this are the autonomous agents that have to either work as a team (cooperative) or against each other (competitive) in a shared dynamic environment of which they do not know the entirety of its state in order to reach any individual or common goal. To achieve this they first must be able to communicate, in a structured way, and to act accordingly to achieve their goals. A lot of literature, researchers and entities are dedicated to this subject and all of its intricacies so we are to expect further progressions within the field in the upcoming years.

The RoboCup initiative is a great way to improve both these fields - DAI and robotics - and more specifically the research in agent coordination and strategy. In the next chapter the topic of the RoboCup initiative as well as a domain analysis of the same will be exposed in order to better understand the main focus point of this dissertation's development.

This chapter did not intend to perform an in-depth literature review on the topics of autonomous agents and multi-agent systems, but to present some key concepts of the two areas for a better understanding of the main characteristics of the systems that will be further discussed throughout this work.

In the next chapter, some major concepts of Graphical Computation are presented as well as some of its history and modern tools.

## Chapter 3

# Graphical Computation

The term “computer graphics” refers to anything involved in the creation or manipulation of images on computer, including animated images. It is a very broad field, and one in which changes and advances seem to come at a dizzying pace [13]. As such, this chapter aims to introduce the most relevant terminology and concepts on the subject that will be used throughout this dissertation’s development with a more substantial emphasis on three-dimensional environments.

### 3.1 Brief History

A generalized consensus tell us that the first computer to have graphical visualization resources was the Whirlwind, developed by the MIT (Massachusetts Institute of Technology) in the early 50s. Its main capabilities were of visualizing numeric data on screen and not the way we today have of computer graphics [33].

Ivan Sutherland, an American scientist and internet pioneer considered to be "the father of computer graphics", was responsible for a huge leap in the computer graphics field with his creation of *Sketchpad* in 1963, during his doctorate’s program. *Sketchpad*’s capabilities were the first to allow object oriented graphics in a sense that object could be created independently of each other and also edited in the same manner, ie. moving a specific vertex of a polygon, both adjacent sides will be moved . All of this was controlled with a "light pen" and a set of push buttons that had several distinct functions such as erasing and moving objects. This was grounding breaking for its time. Ivan can be seen using Sketchpad in 1962 in figure 3.1. His work, namely his concepts of data structuring and interactive graphical computation, came to make *General Motors* invest in the development of what became the first **Computer Aided Design** (CAD) program.

Over the years after this major breakthrough many developments were also made in the field but the main progress was due to the arrival and consequent drop in prices of *workstations* in the 80s and, in the more recent years, as a result of the massive increase in computational power of processing units and the creation of dedicated devices.



Figure 3.1: Ivan Sutherland using Sketchpad in 1962 (from [24])

### 3.2 Pixel - graphics base unit

An image that is represented on a computer screen is made up of pixels, small and squared light emitting parts that nowadays exist in the order of 30 million for the most advanced 8K displays (7680 pixels wide by 4320 pixels tall). Each one is capable of emitting a given color and that number has risen from 2 colors (monochrome) - where each pixel was either on, or off - to over 16 million - where each pixel is defined by 24-bit color. Regardless of the amount of pixels on any given monitor or the color capabilities that each individual one possesses, the color values for all the pixels on the screen are stored in a large block of memory known as a frame buffer. Changing the image on the screen requires changing color values that are stored in the frame buffer. The screen is redrawn many times per second, so that almost immediately after the color values are changed in the frame buffer, the colors of the pixels on the screen will be changed to match, and the displayed image will change. A computer screen used in this way is the basic model of **raster graphics** [13].

For some kind of situations when rendering images on a screen, the best approach is not always based on *raster graphics* as it flattens out the information not retaining much detail about the data and often leading to large amounts processing power to manage and render every pixel on the screen individually to its intended color and position. So another way to represent images is by specifying the basic shapes that it contains, objects like lines, circles, rectangles and triangles. This is the idea that defines **vector graphics**. A vector graphics display would store a display list of lines that should appear on the screen. Since a point on the screen would glow only very briefly after being illuminated by the electron beam, the graphics display would go through the display list over and over, continually redrawing all the lines on the list. To change the image, it would only be necessary to change the contents of the display list [13]. This approach works best for images specified by a small number of geometric shapes, as it can keep the amount of information needed to render it low by only storing information about the position of the pixels that make up the shape. On the other hand, to render the same image through *raster graphics* each



pixel information would have to be stored, generating large amount of processing efforts.

<b>Raster Graphics</b>	<b>Vector Graphics</b>
Pixel-based	Shapes based on mathematical calculations
Raster programs best for editing photos and creating continuous tone images with soft color blends	Vector programs best for creating logos, drawings and illustrations, technical drawings. For images that will be applied to physical products
Do not scale up optimally - Image must be created/scanned at the desired usage size or larger	Can be scaled to any size without losing quality
Large dimensions and detailed images equal large file size	A large dimension vector graphic maintains a small file size
Depending on the complexity of the image, conversion to vector may be time consuming	Can be easily converted to raster
Common formats: jpg, gif, png, tif, bmp, psd, eps and pdfs originating from raster programs	Common formats: ai, cdr, svg, and eps and pdfs originating from vector programs

Table 3.1: Raster vs Vector Graphics (adapted from [51])

Since this dissertation was exclusively developed within a 3D environment, which is the RoboViz monitor, this work does not explore in depth the 2D aspects of computer graphics and goes more in-depth in to the three-dimensional reality and its implementation through the OpenGL API available to the Java language.

Although a 3D graphical environment shares most of the basic implementations and rules of a 2D one, for a more extensive work on the latter please refer to [13] and [11].

### 3.3 OpenGL - 3D Graphics API

The OpenGL is an open-source 3D graphics programming library consisting of a series of callable methods (*application programming interface* or **API**) that can be invoked freely to obtain the desired results. It was introduced in 1992 and has, since then, undergone several versions being now at version 4.7 and in constant development by application developers that continue to innovate and improve the framework. The goal of this section is to introduce basic 3D graphics concepts such as defining and transforming objects and projecting 3D scenes into 2D images.

### 3.3.1 Primitives

In 3D graphics the most common approaches of rendering rely more on *vector graphics* than *raster graphics*. A *scene* is built by creating a list of geometric objects in a three-dimensional space through specification of their vertices coordinates. A "world" object is defined by a set of one or more basic geometric shapes such as points, lines and triangles that together form more complex shapes. This technique is referred to as *geometric modeling*.

Those basic shapes that model an object in OpenGL are referred to as **primitives** and are defined by their vertices, which is nothing more than a point in 3D space defined by its *x*, *y* and *z* coordinates. To draw the most basic shape in 3D modeling, a triangle, we would do as demonstrated in 3.1:

```
1  void triangle() {  
2      glBegin(GL_TRIANGLES);  
3      glVertex3f(0.0, 1.0, 0.0);  
4      glVertex3f(-1.0, -1.0, 0.0);  
5      glVertex3f(1.0, -1.0, 0.0);  
6      glEnd();  
7  }
```

Listing 3.1: Drawing a triangle in OpenGL

Each vertex of a primitive is specified by calling a method of the *glVertex* family. In this case we used *glVertex3f* to specify a point in 3D space with *float* coordinates. In this example, since all three vertices have the *z* parameter set to zero, *glVertex2f* could be called instead to obtain the same results.

All vertices must be specified within the *glBegin* and *glEnd* methods regardless of the primitive and the number of desired vertices and/or shapes within a single call. The *GL\_TRIANGLES* parameter passed onto the *glBegin* method defines the type of primitives that are being defined by the vertices introduced next. For example, to draw more triangles one would only have to continue to define sets of three vertices one after the other and the software would know how to link those together in order.

Among other primitives, the simplest is *GL\_POINTS*, which simply render a point at each defined vertex. Its size can be specified by first setting up the renderer to know its size in pixels by calling the *glPointSize* method with a parameter *size* in pixels. Functions like these are referred to as state-changing functions which alter a state that includes all the settings that affect rendering. Other state-changing examples are the *glEnable* and *glDisable* that called with specific parameters turn on or off desired features of the renderer. Other primitives include *GL\_LINES*, *GL\_QUADS* and *GL\_POLYGON* and even more that are demonstrated in figures 3.2, 3.3 and 3.4 below gathered from [13].

### 3.3.2 3D Coordinates

A coordinate system is a way of assigning numbers to points. In two dimensions, you need a pair of numbers to specify a point [13]. In three dimensions, as the name implies, you need three

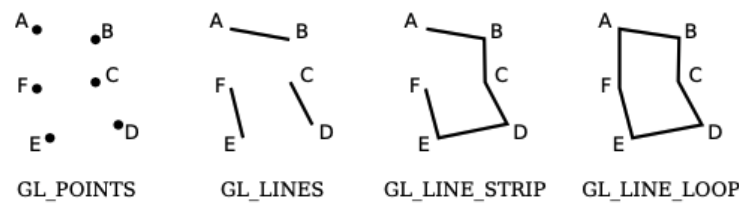


Figure 3.2: Example of line primitives behaviour in OpenGL (from [13])

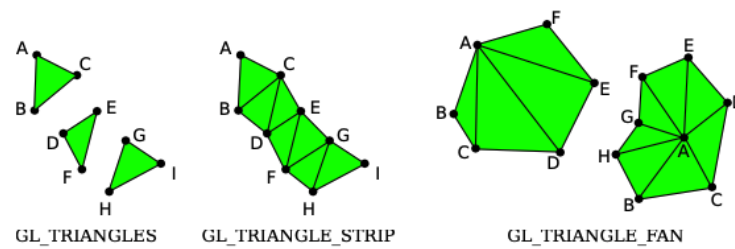


Figure 3.3: Example of triangle primitives behaviour in OpenGL (from [13])

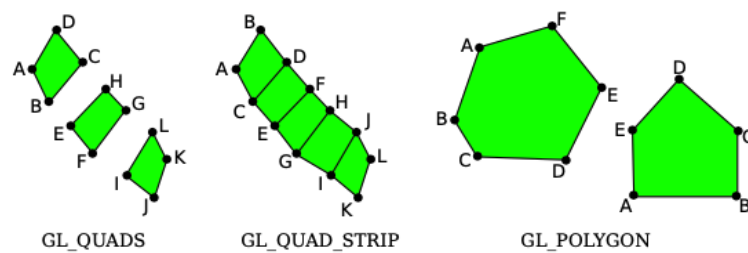


Figure 3.4: Example of quads primitives behaviour in OpenGL (from [13])

numbers to specify a point. Each of those numbers defines the position of the point along each of the axes  $x$ ,  $y$  or  $z$ , normally. These numbers, besides defining points and therefore shapes, make the entire universe of the 3D modeling environment easy to be mathematically manipulated and transformed.

The orientation, direction and name of the axes is arbitrary and, although there is usually some consensus within the same industry, an example of disparity is depicted in figure 3.5 where two different pieces of software organize their coordinate system in distinct manner. When this is the case and resources are shared among the two applications, conversions may need to be applied and the process of axis reorganization is through the use of transforms, the next topic that will be discussed.

### 3.3.3 3D Transforms

Besides aiding in the transformation of one coordinate system to another of different configuration, geometric transformations can also be used to place graphics objects into a coordinate system

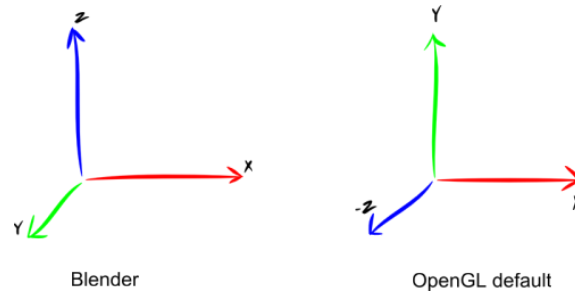


Figure 3.5: Differences in Axis orientations in both Blender and OpenGL (from [39])

[13]. The most basic application of this principle and that is present in each and every graphical environment is the transformation of *world coordinates* to *screen coordinates*. This process is at its core a coordinate transform that takes each object's coordinates within the scene and maps them to pixel coordinates in the viewer's screen viewport. However, within the modeling environment itself, when we want to place the object into a scene, we need to transform the object coordinates that we used to define the object into the world coordinate system that we are using for the scene. The transformation that we need to achieve the desired results is called a **modeling transformation** [13] and in 3D environments is achieved through matrices multiplications between 3D points  $(x, y, z)$  and each respective transformation matrix, as we will see in the examples below.

### 3.3.3.1 Translation

The translation transform is the simplest and, as its name implies, it translates the scene object to a new set of coordinates along the axes, each defined by a number that indicates the amount of motion in the direction of each of the  $x$ ,  $y$  and  $z$  axes that is then multiplied by the point itself to get the final coordinates:

$$P' = T.P$$

where  $P'$  represents the resulting point,  $T$  is the transformation matrix and  $P$  is the original point itself.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

In OpenGL, a translation would be applied by the command family *glTranslate* that allows the specification of its parameters' type as before through appending the type initial letter to the call, ie. to translate a scene object the *float* amounts of  $(t_x, t_y, t_z)$  along each axis:

$$glTranslatef(t_x, t_y, t_z);$$

### 3.3.3.2 Rotation

Regarding the rotation of object in a 3D environment, its application is not as simple as in a two-dimensional world. In three dimensions there are three axis upon which an object can be rotate in and, as such, it is necessary to specify the desired rotation axis or axes and each is applied by their unique matrix multiplication. Therefore, the 3D point matrix representation is given by

$$P' = R_a(\theta).P$$

where  $R_a$  is the defining matrix of rotation for each axis  $x$ ,  $y$  and  $z$  which are described bellow.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In OpenGL, the rotation logic for each independent axis is abstracted and to apply it to a given object we call a member of the *glRotate* family that, as previously discussed allows for the specification of its parameters' type. These methods however have an extra parameter that specifies, in degrees the angle,  $\theta$ , of rotation. The remaining three parameters specify the axis of rotation, which is the line from (0,0,0) to (ax,ay,az).

$$glRotatef(30, 1.5, 2, -3);$$

### 3.3.3.3 Scaling

The process of scaling an object simplifies again the multiplications. The goal of scaling an object is to expand or compress its dimensions across any given axis. The final points are obtain by pure multiplication:

$$P' = S.P$$

However, there are two forms of scaling objects in a scene and each produces distinct results.

**Scaling about origin** changes the size of the object and re-positions it relative to the coordinate system's origin, therefore, the final center position of the object will not be the same after a scaling about origin is applied.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

**Scaling about a fixed point** is a more complex process that consists of three steps:

1. Translate the object so that the fixed point coincides with the origin.
2. Scale the object with respect to the origin.
3. Use the inverse translation of the first step to return the object to its original position.

This procedure can be described into the following operation where  $T$  is the arbitrary fixed point to translate about and  $S$  is the scaling amounts of each axis

$$T(x_f, y_f, z_f) \cdot S(s_x, s_y, s_z) \cdot T(-x_f, -y_f, -z_f)$$

then the resulting corresponding composite transformation matrix would be

$$S = \begin{bmatrix} s_x & 0 & 0 & (1-s)x_f \\ 0 & s_y & 0 & (1-s)y_f \\ 0 & 0 & s_z & (1-s)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In OpenGL, the scaling transformations follow the same logic as before and the functions' family is *glScale*. They each take three parameters that specify the scaling amount on each individual axis and, by definition of matrix multiplication, to leave any axis unchanged, default the value to 1.

*glScalef*(2, 1, 0.5);

The API does not expose a method for scaling over a fixed point so, in order to achieve those type of results, the previously described three-step process has to be applied with use of the aforementioned translation transformations.

### 3.3.4 Hierarchical Modeling

Modeling transformations are often used in hierarchical modeling, which allows complex objects to be built up out of simpler objects. In hierarchical modeling, an object can be defined in its own natural coordinate system, usually using (0,0,0) as a reference point [13]. After modeling the desired object, it can be translated, scaled and rotated into any pose in order to stand by its own or to be part of a more complex object. For this modular technique to be implemented it is necessary a form of limiting the effects of any given modeling transformation and the most common approach is by implementing a stack of transform matrices. Before beginning to model an object, we can push a copy of the current transform matrix onto the stack. When finished, restore the previous transform by popping it from the stack. This procedure can be repeated indefinitely to obtain the most complex scenes although, at some point, the transform's management can get complicated.

In OpenGL, stack management is easily achieved through two basic method calls that, respectively, push or pop the current transform matrix, which is a composition of all the transformations applied up to that point, onto or out of the transform stack.

`glPushMatrix();`

`glPopMatrix();`

A great and simple example of hierarchical modeling and how it can be used to achieve more complex objects through the transformation of existing, simpler objects, can be found in David J. Eck's book *"Introduction to Computer Graphics"* which was implemented in OpenGL version 1.1 but that still applies to more recent versions. First, we can define a function, see 3.2, that draws a simple shape that can be reused. In this case, a square of a given color defined by its *r*, *g* and *b* parameters.

```
1 void square( float r, float g, float b ) {  
2     glColor3f(r,g,b);  
3     glBegin(GL_TRIANGLE_FAN);  
4     glVertex3f(-0.5, -0.5, 0.5);  
5     glVertex3f(0.5, -0.5, 0.5);  
6     glVertex3f(0.5, 0.5, 0.5);  
7     glVertex3f(-0.5, 0.5, 0.5);  
8     glEnd();  
9 }
```

Listing 3.2: Drawing a square in OpenGL

To draw a cube, a total of six faces would be needed, therefore, the *square()* function would be called six times, one for each of the faces. But since the function defines always the same shape built by the same vertices coordinates, we need to alter the current transform so that we can render the face at a different position. In order to not modify the previous transforms applied to the current object or scene, we must manipulate the transform stack, as previously explained (see 3.3).

```
1 glPushMatrix();  
2 glRotatef(90, 0, 1, 0);  
3 square(0, 1, 0);  
4 glPopMatrix();
```

Listing 3.3: Modifying the transform stack

The end result, after calling *square(1,0,0)*, applying the stack modifications and finally re-calling *square(0,1,0)* would render a two-faced cube centered at the origin of the coordinate system (see figure 3.6).

In order to render the rest of the faces we would only need to apply the remaining stack transformations necessary to modify the current transform matrix and call the *square()* function at each point. In his book, Eck defines the function to do that procedure (see 3.4) and that takes in a parameter *size* that takes advantage of the scaling transforms to define the final cube's size.

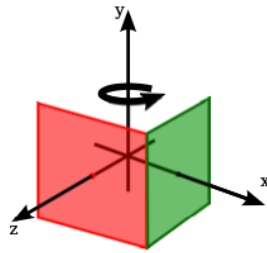


Figure 3.6: Example after rendering two faces (from [13])

```

1  void cube(float size) { // draws a cube with side length = size
2      glPushMatrix(); // Save a copy of the current matrix.
3      glScalef(size, size, size); // scale unit cube to desired size
4      square(1, 0, 0); // red front face
5      glPushMatrix();
6      glRotatef(90, 0, 1, 0);
7      square(0, 1, 0); // green right face
8      glPopMatrix(); glPushMatrix();
9      glRotatef(-90, 1, 0, 0);
10     square(0, 0, 1); // blue top face
11     glPopMatrix(); glPushMatrix();
12     glRotatef(180, 0, 1, 0);
13     square(0, 1, 1); // cyan back face
14     glPopMatrix(); glPushMatrix();
15     glRotatef(-90, 0, 1, 0);
16     square(1, 0, 1); // magenta left face
17     glPopMatrix(); glPushMatrix();
18     glRotatef(90, 1, 0, 0);
19     square(1, 1, 0); // yellow bottom face
20     glPopMatrix();
21     glPopMatrix(); // Restore matrix to its state before cube() was called.
22 }

```

Listing 3.4: Rendering a full cube in OpenGL

### 3.3.5 Projection and Viewing

Between the local transformations within a given object - **modeling transform** - to the final device coordinates where the rendered image is displayed - **viewport transform** - several other transforms and calculations take place in order to normalize the myriad of transformations and obtain a final render-ready result based on the position of the "viewer" - the camera. This is a complex process (see figure 3.7) that is outside of the scope of work of this dissertation, please refer to [44] and [13] for a more in-depth explanation.



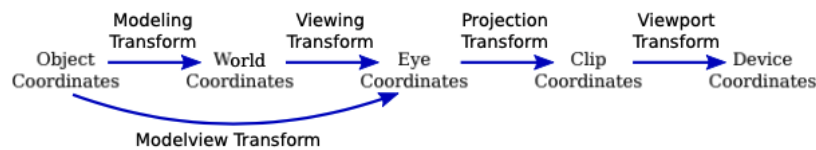


Figure 3.7: Viewing transformation process (from [13])

### 3.3.5.1 Virtual Camera

Projection and viewing are often discussed using the analogy of a camera. A real camera is used to take a picture of a 3D world. For 3D graphics, it is useful to imagine using a virtual camera to do the same thing [13]. The virtual camera is an important concept in computer graphics, as it represents the point of view on which objects belonging to the virtual world are projected on the screen. By definition, the crucial part of any virtual environment, as it is vivid, depends largely on the camera's view, as it is one of the primary means by which information from the virtual environment is transferred to the user via the screen [33].

In general, seven degrees of freedom by which a virtual camera is controlled are considered: 3 degrees to move in the Cartesian graphic plane ( $x, y, z$ ), 3 degrees to rotate (pan, tilt and roll) and 1 degree to change the field of view (zoom).

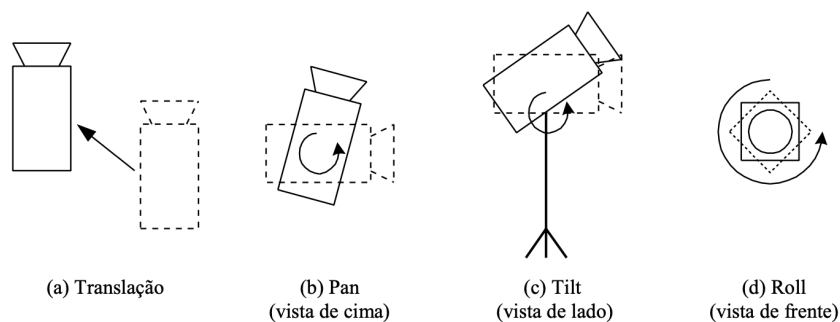


Figure 3.8: Possible Virtual Camera Movements (from [33])

For positioning the camera in the virtual environment, it is always necessary to use the following camera parameters [52]:

- **Camera Positioning  $P(x,y,z)$ :** Point at which the virtual camera is placed in the three-dimensional coordinate system.
- **Direction Vector  $A_r(\partial x, \partial y, \partial z)$ .**
- **Field of View:** The horizontal and vertical field of the viewing angles, called the "lens angle". In computer graphics, this angle is determined through two parameters, the horizontal field of viewing angle (H) and the vertical field of viewing angle ( $\theta V$ ).

- **Up Vector of orientation  $U_p(\partial x, \partial y, \partial z)$ :** Vector that controls the angle of the camera's tilt or roll rotation over its direction vector.

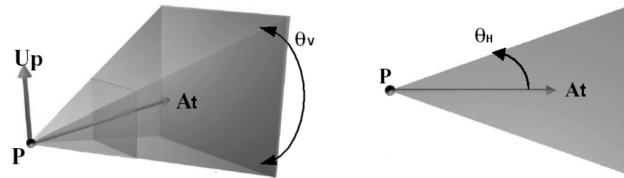


Figure 3.9: Virtual 3D Camera Parameters (from [33])

### 3.3.6 3D Visualization Pipeline

The 3D visualization pipeline is the basic process of generating a two-dimensional projection from a three-dimensional scene taking into account its lighting and viewer's perspective so that it can then be rendered to a screen. The process takes as inputs a set of polygon-based objects of a three-dimensional scene, its light sources and visualization parameters and outputs the information regarding the final visible pixels on a display device.

Its is, again, a complex process but that can be summarized in the following ordered steps that describe the PBR (Polygon-Based-Rendering) procedure [54]:

#### Transformation to Global World Coordinates

- To compose a scene in 3D space consisting of different objects, all created 3D objects must be transformed into the same coordinate system.

#### Transformation to 3D View Coordinate Systems

- A viewpoint in 3D space is cited as the "camera" location.
- The geometry from the 3D space is transformed into the camera view coordinate system. The projection from 3D to 2D space is performed at this stage.
- The depth information of any object can be obtained from the  $z$  coordinate value.
- The effect of virtual "lights" that creates illumination properties in the 3D scene is computed.
- The removal of polygonal surfaces not shown in the view due to occlusion is known as "culling" and is performed at this stage as well.

#### Transformation to 3D Clip Coordinate System

- The geometry data in this stage are prepared for a post-processing step known as "clipping".

**Transformation to Normalized Device Coordinates**

- The geometry is normalized for a display in a 2D window on a physical display device.
- Further clipping is done to remove geometry outside of the user-defined window boundaries.

**Transformation to Display Window Coordinates**

- All vertices are converted to units of the display (pixels) window.

**Transformation to 2D Screen Coordinate System**

- The conversion to screen pixels (rasterization) is performed.
- The output of this stage is the final color of every pixel placed in the memory of the display hardware (the frame buffer).

### **3.4 Conclusions**

Computer graphics, whether two or three dimensional are a complex and extensive process that requires heavy computational power to be able to crunch the numbers of the many underlying necessary mathematical calculations that compose a final 2D rendered image or a real-time three dimensional interactive environment. Much so in the current times of 4K and 8K displays with millions of pixels and the never-ending requests for better and more realistic simulations. Fortunately, modern tools provide a great level of abstraction from these procedures enabling application developers to break through the boundaries of imagination and realism at a faster pace than ever. Like many others, this is a major field that I look forward to seeing the new advancements made into the future.



## Chapter 4

# Cinematography: The Art of Visual Storytelling

Cinematography is the art of making films. The filmmaker's central problem is how to capture for the shoot a set of events that take place in the world simultaneously so that the viewer can perceive them with all clarity, giving the intended continuity [33]. This chapter is short and intended only to give a baseline understanding of the concepts of camera manipulation and storytelling. For further references, please refer to [25] and, for a more game-directed approach, [8].

### 4.1 The Principles of Cinematography

The main principles of cinematography were not set at a single moment in time, they were created, applied and developed through time and with the progresses of the industry. They first ones have, however, for the most part, remained intact and a source of truth for modern techniques. The main aspects to take into consideration when shooting a set of actions through a camera can be ruled by the following basic concepts [27]:

**Camera Angles** - Choice of perspective and camera angle in shooting narrative film can be motivated by many reasons. For example it can be following a subject, revealing or withholding information, providing graphic variety or setting a specific mood.

**Framings** - Open and closed framings determine if the viewer is included or excluded from the picture span. Open framing is when the object and situation within the picture space is not set and positioned for best clarity before filming. Open framings can often be seen in documentaries for example. Closed framing is when subjects are positioned with care for best graphical balance. Open framings appears more realistic for the viewer as closed framings seem more staged and controlled. item

**Point of View** - In different framings, the viewer's different levels of involvement are determined. Point of view on the other hand determines with whom the viewer' involves and

identifies with. The importance of point of view is that it decides the way a viewer interprets a scene. Different narrations of point of view are used in film, but the most common are first-person point of view, third-person restricted point of view, and omniscient point of view.

**Camera Movement** - When a moving shot replaces a series of edited shots, it creates a rhythmic variation and realistic simulation for the viewer. There are three main camera movements: panning, craning and tracking. Panning is when a camera follows the object, without ever moving out of position. Craning is when the camera is put on a crane and therefore can perform a variety of different framings and difficult variations in one shot, such as high shot, low shot, open and closed framings. Tracking is a shot following a subject, with a moving camera.

The relationship between game cinematography and its traditional counterpart is extremely tight as, in both cases, the aim of cinematography is to control the viewer's perspective and affect his or her perception of the events represented [8]. However, as Burelli explains in his work, game cinematography differs from the traditional aspects of film-making in a sense that a big part of game actions are not pre-scripted events that are expected to occur in a controlled environment with all known variables, so camera control gains another dimension, that is reaction.

## **4.2 The Camera**

The main dilemma of a director is to decide where to place the cameras and lights so that the captured scenes are able to describe an uniform and coherent set of events. Undoubtedly, in virtual environments camera placement and parameters settings are much easier to achieve due to clear advantages to not having physical restrictions like space and uncontrollable environment variables.

### **4.2.1 Virtual Camera vs Real Camera**

The virtual camera is an important concept in computer graphics, as it represents the point of view on which the objects of the world are projected on the screen. A virtual camera can be placed anywhere in the graphic world, including inside solid objects. In the virtual world, cameras do not have any limit of movement from one point to another or rotation around any axis, and this movement can occur at an instantaneous speed [33].

A real camera, on the other hand, has a set of physical limitations. It cannot penetrate solid objects and it cannot fly around an object just as easily. Despite the limitations of real cameras, it is important to impose some restrictions on virtual cameras so that they resemble reality.

### 4.2.2 Physical Restrictions of the Camera

Physical restrictions can also be applied to the virtual camera so that it becomes as similar as possible with a real camera in its limitations. The limitations to be established in a virtual camera are [33]:

**Camera rotation speed** limits the speed of the virtual camera rotation.

**Camera movement speed** limits the movement range of the virtual camera. For example, in a scenario with multiple cameras, it should be ensured that moving to new positions is possible with a real camera.

**Zoom speed** restrictions limit the speed of the camera to change the zoom level, while also limiting the maximum and minimum zoom levels.

The inclusion of these three physical restrictions, thus covers the 7 degrees of freedom of a virtual camera in your three-dimensional world.

### 4.2.3 Intelligent Camera Control

In this area, Sérgio Louro[33] does research on work related to the problem in planning the control of film cameras. In it, he evaluates the following approaches:

1. **Jim Blinn** - Spacial System [5]
2. **Gleicher** - Assistant for automatic camera control [20]
3. **CamDroid** - Intelligent camera control system [42]
4. **ConstraintCam** - Camera control through restrictions [4]

In the end, the main agreed on restrictions for a successful and realistic camera control are the following:

1. **Look At Point:** restricts the camera's focus to a specific point (object);
2. **Object in Field of View:** restricts that an object is in the field of view;
3. **Object Occlusion Minimize:** restriction of objects with a certain value minimal occlusion. Occlusion values range from [0.0 to 1.0];
4. **Object View Angle:** restriction of the camera's orientation angle with respect to the object. The orientation is expressed in spherical coordinates;
5. **Object Distance:** restriction of the distance from the camera to the center of the object;

6. **Object Projection Size:** restricts the distance from the camera to the object and the angle of the field of view determines the size at which the object is viewed, using for example for a close-up;
7. **Object Projection Absolute:** Require the projection of a main object to be completely within a certain rectangular region of the image in two dimensions.
8. **Object Projection Relative:** The projection of the main object stipulates a relationship for the projection of the secondary object in the image.
9. **Object Depth Order:** restriction to specify that the camera is positioned so that the pivot object appears as close or far to the camera as a secondary object.
10. **Camera Position Region:** restricts the region of action of the camera in three-dimensional space;
11. **Camera Field of View:** restricts the angle of the camera's field of view vertically and horizontally to define the format of the image to be viewed.

### 4.3 Conclusions

The art of conveying a compelling story to a viewer through a lens is not as straightforward as one may initially assume. However, there are two major distinct areas of story-telling, scripted - *a movie or TV series* - and non-scripted - *live events*. The approaches to accurately portray each of these are definitely farther from similar but, most certainly, much of it is reliant on good camera positioning and manipulation with regards to the same solid principles of the craft. For this work, this research will have a great impact on the development of the cameras orchestration module and how to better depict the idea of a realistic soccer simulation broadcast.

In the next chapter, a brief overview of the history of soccer is made and RoboCup initiative is introduced further with a larger emphasis on the RoboCup Soccer Simulation League.



## Chapter 5

# Domain Application Analysis: Human and Robotic Soccer

Soccer (aka Football in most countries) is considered to be the *King Sport* all around the globe. Although some countries have their own distinct most popular sport, the global presence of soccer is much higher. In the graphic 5.1, represented by the green color, we can see which places soccer dominates as the most cheered sport. In virtually all of Europe, South America, Africa, and the Middle East, soccer is king, making up the majority of the worldwide sporting interests.

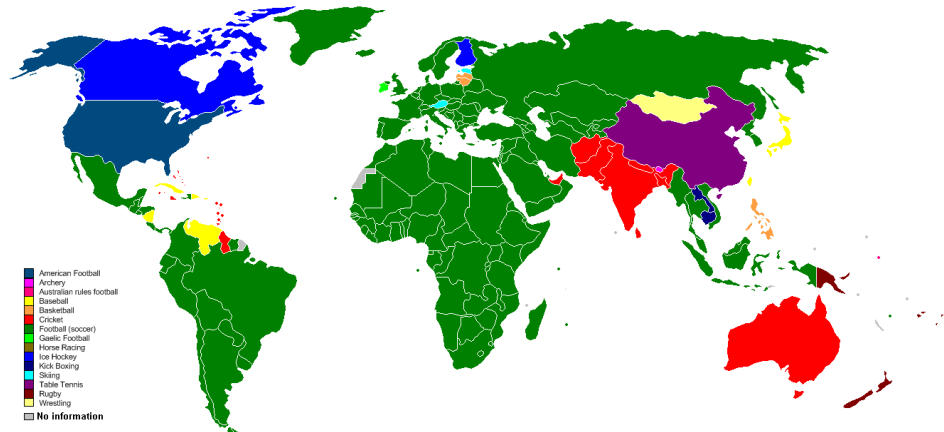


Figure 5.1: Soccer popularity around the world (from [1])

Apart from the large affluence of fans in sheer numbers, soccer also became one of the largest businesses in the sporting industry with the European market alone generating over €28 billion of revenue annually and with players being paid salaries of above €65 million.

As stated before at the introductory stage of this dissertation, soccer, being the major influencing sport worldwide, also served as ground for many research initiatives that help to evolve various technological fields and, in this chapter, we will briefly discuss the origins and evolution of the sport through the years as well as one of the aforementioned initiatives that has been adopted globally, the RoboCup international competition.

## 5.1 Human Soccer

### 5.1.1 History

Soccer has a long history. In its current base form, two teams with same number of players have a common objective to navigate a round object into the net of the opposing team. But the sport as we know it today is simply the end result of a slow and steady evolution of different ball games throughout history.

According to some authors [37] [58], several versions of the game of soccer have existed in ancient times: *pok-a-tok* (Mesoamerica), *tsu chu* (Ancient China), *Kemari* (Japan), *Epyskiros* (Ancient Greece), *Haspartum* (Ancient Rome), Gaelic football (Ireland), *Soule* or *Choule* (France), *Calcio* (Italy) and others:

- **Kemari (Japan)** - Two teams played, each one consisting of eight players trying to keep the ball in the air using only their feet. The ball was full of sawdust wrapped in deer leather. The field of the game, named *kikutsubo*, was rectangular [58].



Figure 5.2: *Kemari* illustration by Akisato Ritoh 1799 (from [31])

- **Pok-a-tok (Mesoamerica)** - The only available information about *pok-a-tok* comes from wall and religious paintings, as well as from preserved game fields, where this game was played. The oldest game field of *pok-a-tok* dates back from 1600 B.C. This field has the shape of the capital letter I. At the two opposite sides of the field, there were parallel walls nine meters in height. On each of these walls, there were either three hanging saucers or an engraving ring. The players had to kick an elastic ball no more than 15 cm in diameter on the saucer or in the ring. The ball could be kicked either by knees, hips or elbow [58].
- **Tsu chu (China)** - According to historical records, aristocrats, soldiers and folks used to play this special kind of kicking game from 2500 B.C. or even earlier. The game's name

comes from the word *tsu* that means “the ball which is full” and the word *chu* that means “kicking the ball with a foot”. The aim of this game was to kick the ball so as to place it in a hole of approximately 30–40 cm in diameter. The hole was formed by a net hanging between two bamboo sticks that were nine meters above the ground. The players used exclusively their feet and the “goal” was extremely difficult considering the small diameter and the height of the target [58].

- **Episkyros (Greece)** - The ancient Greek *episkyros* was played around 2000 B.C. In particular, there is a marble relief in the National Museum of Archeology in Athens, which shows an athlete balancing a ball on his thigh. Some historians believe that the athlete demonstrates the football technique of *episkyros* or *ephebike* or *phaininda* to a boy. In *onomasticon* there was a vague description of *episkyros*. According to the dictionary, two teams of equal number played *episkyros* and the players of both teams inscribed a line on the ground, which was called *skirus*. This line split the two teams. Players were throwing the ball in order to pass the opponents’ “goalpost”.

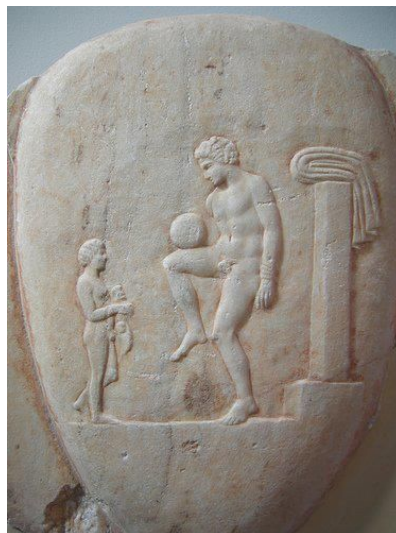


Figure 5.3: Stone carving that shows a man balancing a ball (from [3])

- **Haspartum (Rome)** - There is no written evidence about the way *harpastum* was played. As far as we know, two teams played on a rectangular field split by a centerline. Each team had to keep the ball in its own area for as long as it could, while its opponent tried to steal it and get it over to its own side. It seems that it was a difficult and rough game and for this reason soldiers played it in order to maintain their physical fitness [58].
- **Gaelic football (Ireland)** - Gaelic football is similar to the contemporary soccer. Its history begins in Ireland in the Middle Ages. Gaelic football or Irish *peil gaelach* or *caid* is one of the four traditional Irish games, along with *camogie*, which is hurling played by women, Gaelic handball or Irish *lianhtróid* which looks like contemporary handball and rounders or Irish *cluiche corr* which looks like the contemporary softball [58].

- **Soule or Choule (France)** - During the medieval times, a game named *choule* or *la soule* was played by the European nobility, especially by the French, on Sundays or on feast days. The aim of the game was to pass a ball to the opponents' goalpost, which was a tree, a wall or a rivulet. The ball symbolized the sun. It was a rough and violent game without definite rules about the use of the hands and the legs [58].
- **Calcio (Italy)** - A game named *calcio* or *giuoco del calcio forentino* was played in Italy in the 16th century A.D. . It was played exclusively by aristocrats, every night during the period of the Epiphany and the Lent, in most of the towns north to the Rome. Using their hands and legs two teams of twenty-seven players were trying to pass a ball to the opponents' goalpost, which was a target in the perimeter of a field [58]. Italian football still retains the original name of the sport in the name of its 3rd division, *Lega Italiana Calcio Professionistico*, aka *Serie C*.
- **Foot-ball (England)** - Said to be the source of the sport that we know and love today, in England, at the beginning of the 19th century, the game began to spread immensely, leading to the creation of the first set of rules of the game through the establishment of the Football Association in 1863. Its main goals were to organize matches between several teams country-wide and to normalize the rules of the sport across all of the countries' regions [12]. In 1888 the first soccer league was created, the England Soccer League and shortly after, in 1904, FIFA (Fédération Internationale de Football Association) was constituted with the main goal of normalizing the matches' rules and organizations.

### 5.1.2 Characteristics

Castelo [10] defends that "The game possesses a specific dynamic, a context that defines its essence. This essence, included in the game rules, gives rise to a series of attitudes and technical/-tactical behaviour patterns. More specifically, the requirements that are imposed on the players are determined by the game profile". In consequence of that, in 1982, Kacani divided players into three categories:

- **Universal Players** - Players are capable of fulfilling with the same performance many tasks, both in the offensive and defensive field.
- **Semi-universal Players** - Players are capable only to perform optimally in one of the game key moments (offensive or defensive).
- **Specialists** - Players with a defined expertise, able to perform effectively in a particular sector of the field (.ie the goalkeeper).

The sport in it self, since its initial creation, has gone under several evolutions, the transformation from a more individual aspect to a more collective one, the game's systems evolution and the emergence of players with more excellent technical skills and, finally, the third evolution consists in using the better soccer concept between tactical-physical and tactical-technical dimensions [12].

### 5.1.3 Tactical Evolution

In the beginning there was chaos, and football was without form. Then came the Victorians, who codified it, and after them the theorists, who analysed it [53]. As of now, it is common sense that the arrangement of the players on the pitch is directly related to the team's performance and result. In the beginning of the practice of the sport today known as soccer however, there was not much sophistication around this matter and the game played out without much organization or structure.

It wasn't until the late 1920s that player's positioning on the pitch began to approximate what we see today in the professional leagues around the world. In the years prior, a lot of experimentation and adjustments were made to allow for the teams to go from a "chase the ball" mentality to one of "united action" where passing and positioning became more relevant.

In beginning, formation was reliant primarily on offense, the main tactic was to chase the ball and put it in the back of the net as many times as possible, taking advantage of any advantages available like body weight or faster pace. As such, the formations at the start of the era of organized matches was something on the line of what is depicted on figure 5.4 a). Scotland was the clear underdog at this match-up, but, through a more mid-field balanced tactical choice, they were able to counter-balance England's team and finish the game with a goalless scoreboard.

In the years that came, the strategies had evolved what would later be described in the 1880s as: the Pyramid. As seen in figure 5.4 b), the teams took different approaches and the pyramid took home the victory. The Wrexhams balanced out the offense and reinforced their midfield ultimately creating a stronger core that led them triumph.

As the successes of a balanced distribution of players through the entire pitch were increasing, more and more teams began implementing their own versions of the scheme. Besides the team's formation it was notorious that the individual skill began to gain relevance in order for the team as a whole to be able to out-perform its equally well-organized opponent. Figure 5.4 c) depicts two strongly-organized teams with much individual talent on what came to be a greatly disputed match.

The previous examples were extracted from Jonathan Wilson's book *"Inverting the Pyramid - The History of Football Tactics"*, where he explains in depth the evolution of tactics and formations since the 1880s. Please refer to his work, [53], for more information.

## 5.2 Robotic Soccer - RoboCup

The RoboCup initiative [22] [23] is an international research and education project which main objective is to promote investigation in (Distributed) Artificial Intelligence and Intelligent Robotics. The base research problem of this project is Robotic Soccer, where a large number of technologies and methodologies is necessary to be able to create and manage a real or virtual team robotic team that has the ability of playing a soccer game while abiding to a set of distinct rules.

The initiative was first born in Tokyo, where a group of researchers promoted a workshop related to the use of soccer for the research community, specially in AI areas. Some robotic soccer

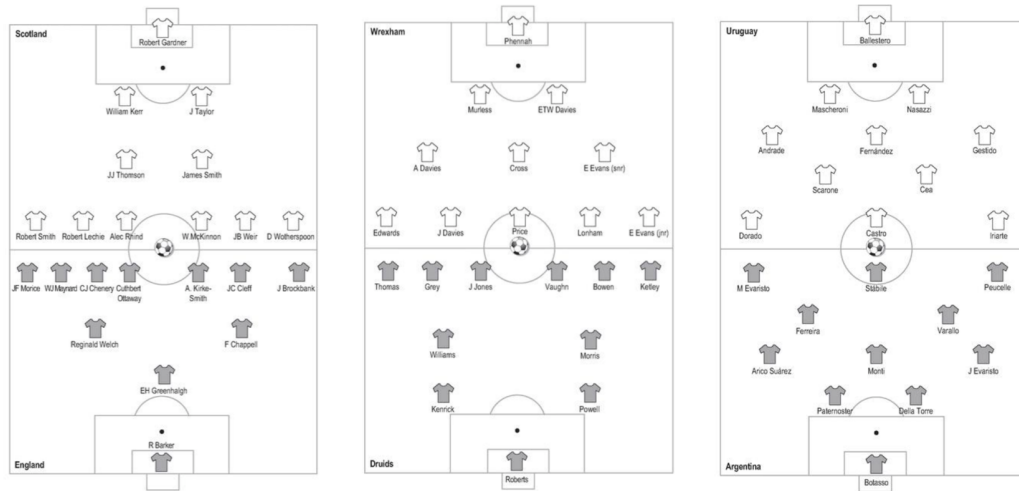


Figure 5.4: a) First International: Scotland 0 England 0; b) Wrexham 1 Druids 0; c) Uruguay 4 Argentina 2 (from [53])

prototypes and a simulator project were defined and the result was the creation of a Robotic League called Robotic J-League. After its huge success, this project became international with the name of Robotic World Cup Initiative - RoboCup [12].

In order to promote research in this field and stimulate development, a long-term goal was set:

*"by the year 2050 a humanoid Robotic team will be capable of defeating the world champion Human team in a soccer match according to FIFA rules"*[23]

The first RoboCup competition was held in Nagoya in 1997 with over 40 teams. The organization estimates that over 5 million spectators assisted the games, which turned the RoboCup into one of the biggest events ever [12]. Now, the RoboCup events gather upwards of 3500 researchers with their 3000 robots to compete in its various leagues of specific areas of research. Apart from the main floor where the competitions take place, at each RoboCup event there is also the occasion for an International Symposium, a high place for presentation and discussion of scientific contributions. This meeting of researchers allows to highlight the latest advances in robotics and artificial intelligence with the best researchers in the subject [2].

## 5.2.1 RoboCup Leagues

The challenge proposed by the RoboCup organization for the AI and Robotic researchers spans across 5 different categories: Soccer, Rescue, @Home, Industry and the Junior category. Within each category there are multiple sub-categories that branch out the research focus to even more areas, see figure 5.5.



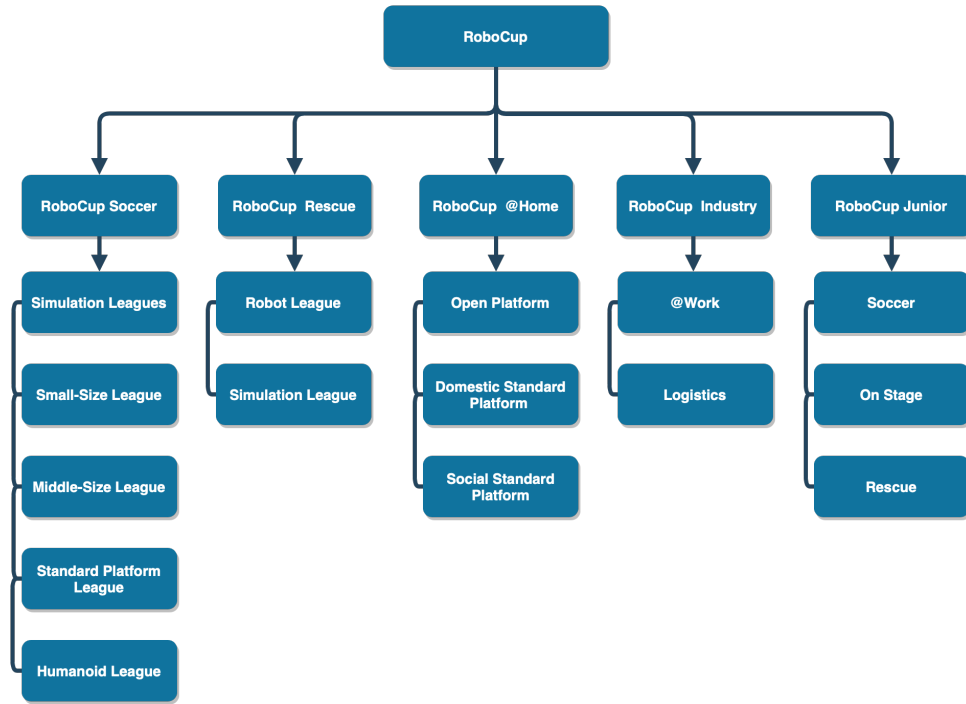


Figure 5.5: RoboCup Initiatives Structure

### 5.2.1.1 RoboCup Soccer

The main event of the RoboCup initiative and the focus of this work is the robotic soccer, where two teams of autonomous and collaborative robots develop dynamic strategies to challenge each other and win the game [2]. This competition is divided into five leagues, including 2 simulation leagues and four main robotic leagues with distinct rules:

#### 5.2.1.1.1 Simulation Leagues

The Simulation League is one of the oldest leagues of the RoboCup initiative and, in it, only virtual robots are allowed. Using the SimSpark soccer simulation servers, SoccerServer, two virtual teams of 11 players simulate a soccer game. These players are simulated agents that use autonomous AI systems capable of interacting with their virtual environment through multiple virtualized sensors that allow them to play strategically as a team in a multi-agent system. Its main focus is on artificial intelligence, team strategy and multi-agent coordination and it is then subdivided into two sub-leagues: **2D** and **3D**. Its first iteration, the 2D Simulation 5.6, was first simulated in 1997 in the Nagoya, Japan tournament and the first 3D server was released in late 2003.

Currently, its official simulation monitor for the 3D sub-league is the RoboViz tool (see figure 5.7), an interactive 3D visualizer that will be the main focus of this dissertation due to its three-dimensional aspect, allowing for a realistic representation of a soccer simulation.



Figure 5.6: 2D Simulation League (from the official website)



Figure 5.7: 3D Simulation League (from the official website)

The 3D Simulation league is a simulation of the NAO robots used in the Standard Platform League and also makes use of color-coded markers to detect distinct flags during the game. With the introduction of the extra dimensional aspect and the use of fully articulated humanoid agents, the main focus of the Simulation League had widened from a simple aspect of team coordination and strategy implementation only, to the low level control of a complex machine that had to be able to walk, kick and stand up after a tackle.

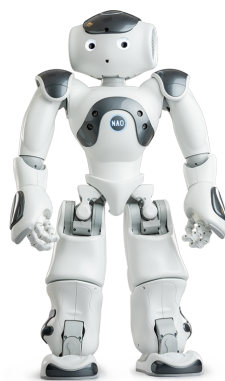


Figure 5.8: The humanoid NAO robot that was modeled for the 3D Leagues



Apart from the increase in complexity compared to the 2D sub-league, this 3D Simulation League also allowed researchers to test out control programs and strategies on the simulated humanoids instead of on a real pitch with expensive resources in place. A normal game has the duration of 10 minutes, split into two 5 minute long halves.

#### 5.2.1.1.2 Small-Size League

On a small pitch, two teams of 8 small sized robots compete using a golf ball. In this league, the robots do not possess individual visual sensors nor are able to make decisions individually. As such, they rely on a central control unit (a computer nearby) and a vision system assembled above the pitch. The control unit receives, analyses and processes information sent by the vision system and computes all necessary parameters - orientation and velocities - to send to each individual unit on their team to, hopefully, win the match. The computing done by the central control unit is the focus point of the researchers who play on this league. Each player unit cannot exceed 18cm in diameter and 15cm in height in order to be allowed at the small-size league. The games have a 10 minute duration and no humans are allowed to interfere.

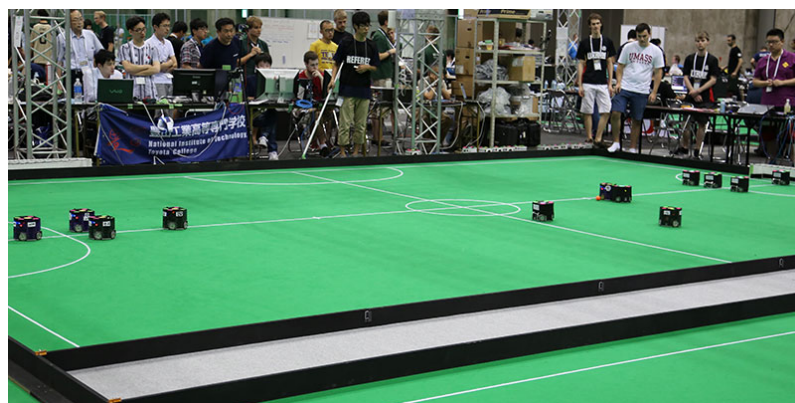


Figure 5.9: Small-size League game (from the official website [2])

#### 5.2.1.1.3 Middle-Size League

Teams composed of 5 to 6 larger autonomous robots compete in a larger, 18 meters by 12 meters field. The players are aware of their local surroundings using their sensors and are allowed to communicate with each other and share their positions on the pitch to structure the team's organization. At this league, a normal indoor-soccer human-sized ball is used and the robots are much more powerful than their smaller counterparts. Each half of the game is 10 minute long, totaling a 20 minute match with the possibility for substitutions.

#### 5.2.1.1.4 Standard Platform League

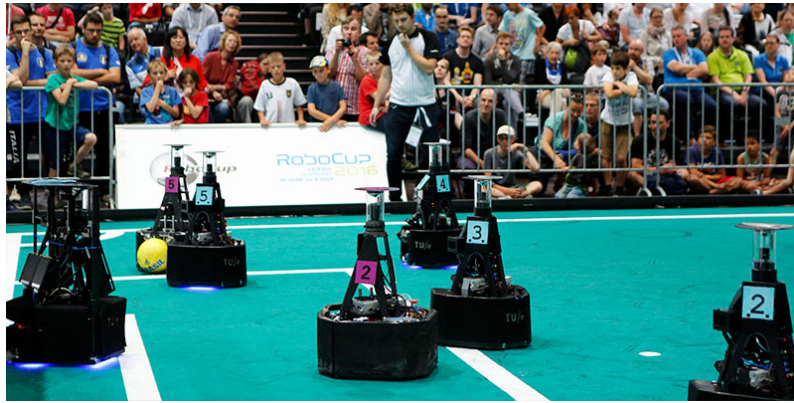


Figure 5.10: Middle-size League game (from the official website [2])

Teams composed of 5 humanoid Softbank Robotics' Aldeberan NAO robots compete in a 6x9 meters pitch. Like the league before, each half has a 10 minute duration and there are several color-coded markers around the pitch so that the robots are able to recognize several flags globally to determine, through their vision sensors, the locations of the pitch lines, the goals, their team-mates, their opponents and, of course, the ball. The robots communicate freely with each other and make their own decisions.

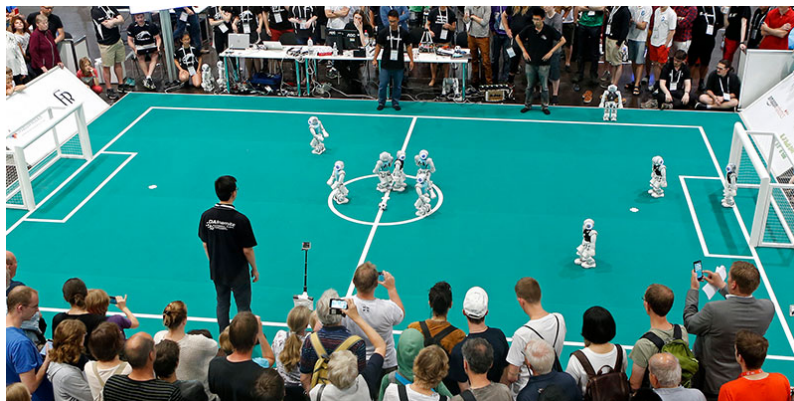


Figure 5.11: Standard Platform League game (from the official website [2])

#### 5.2.1.1.5 Humanoid League

Autonomous robots, freely constructed by participating teams, with a human-like body and senses play soccer against each other. The league is divided further based on robot size creating sub-sub-leagues of soccer, being them *kidsize*, *teensize* and *adultsize*. The number of players varies between 2 and 3 and, with it, the sizes of the pitch, ball and goals. These robots do not rely on color-coded markers spread across the field. Instead, this league aims to push further the development of vision recognition based technologies and dynamic balancing of loads. The match's rules are similar to real-life and the duration is, as before, two 10 minute halves.



Figure 5.12: Humanoid League game (from the official website [2])

### 5.2.1.2 RoboCup Rescue

The Rescue initiative of RoboCup puts robots through series of high-difficulty challenges with the main goal of researching developments in the domain of search and rescue in large disasters. The Rescue competition trials are representative of the main challenges encountered in search and rescue scenarios in urban or natural environments, such as navigating in rough terrain, overcoming obstacles, moving objects, mapping, and team coordination [2]. The competition was created in 2001, 6 years after the earthquake disaster in Kobe city - Japan that victimized 6500 people. It is split into 2 categories: simulation and robotic.

#### 5.2.1.2.1 Robot League

In the Robot League, highly equipped robots search for and rescue victims in disaster-like scenarios. The robots have multiple sensors, ranging from temperature, CO2 and vision that allow them to autonomously advance through the most difficult scenarios. The machines' results are based on the total number of obstacles overcome, people found and number of operators. This particular league leads to big advancements in robot technology and to the construction of incredibly capable machines (see example 5.13).

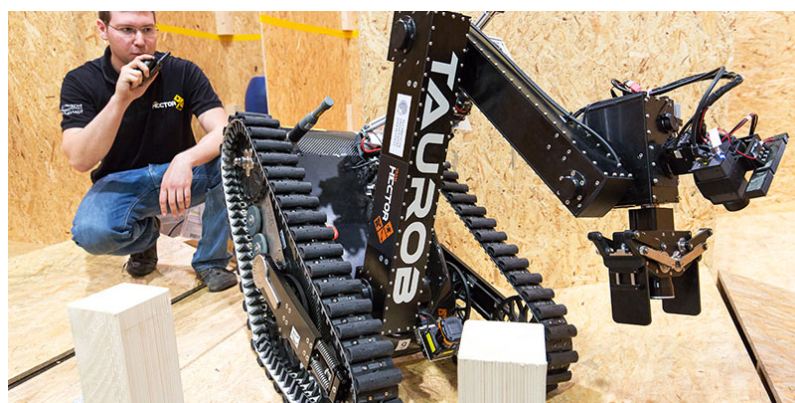


Figure 5.13: Rescue Robot League game (from the official website [2])



#### 5.2.1.2.2 Simulation League

The Simulation League of the RoboCup Rescue initiative maintains all the main goals of the Robot League but instead of an extreme course built to be overcome, it simulates the occurrence of a large disaster on a major urban area. The researchers' teams goal is to deploy rescue, fire-fighting and patrol units in the area to search and rescue civilians in need of help. The problems are not solvable by a single agent and, therefore, the major objectives of this league is to improve cooperation techniques among agents to better work in a dynamic and difficult environment.

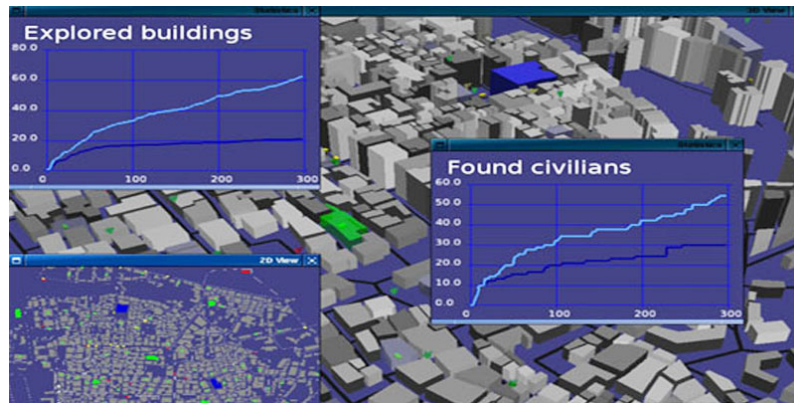


Figure 5.14: Rescue Simulation League simulation (from the official website [2])

#### 5.2.1.3 RoboCup @Home

The @Home initiative aims to develop service and assistive robot technology with high relevance for future personal domestic applications. Mobile, autonomous robots interact with humans to carry out everyday tasks around common living environments like one's home or public spaces like shops. The robots are interactive through gestures, voice commands or any other form that can be developed. It is currently subdivided into three categories: *Open Platform*, *Domestic Standard Platform* and *Social Standard Platform*. Besides the type of robots used to develop solutions in each division, the common goal is to aid is the achievement of mundane tasks interactively.

#### 5.2.1.4 RoboCup Industry

Research application in industrial environments and logistics operations are the main focus. The plan is to help build the factory environments of the future, enabling people to be freed from their boring repetitive tasks at factory floors and at the same time to improve and organize production with robot organization, cooperation and task pooling. These two branches of research serve as a common grounds for the two areas of this initiative: *@Work* and *Logistics*.

##### 5.2.1.4.1 @Work

This division's challenges are based around assistive robots for the industry where their main role is to cooperate with human workers on their daily complex tasks assignments resulting in a safer and more productive working environment.

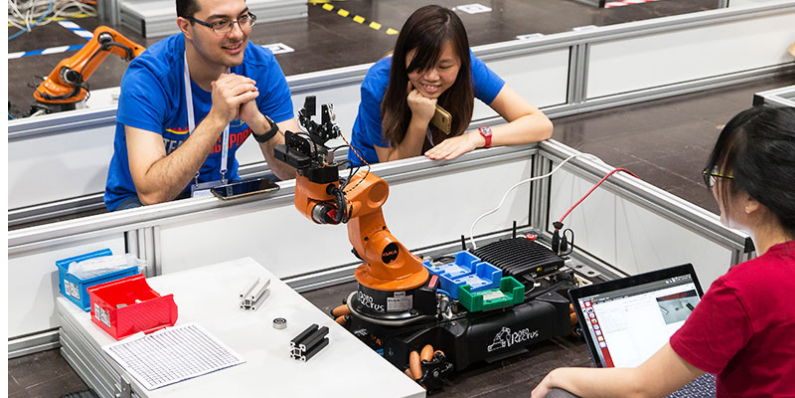


Figure 5.15: RoboCup Industry @Work (from the official website [2])

#### 5.2.1.4.2 Logistics

The logistics category focuses on enabling today's industry to adapt faster to changing product needs and to be more flexible on their production line. The main challenges of this league are to develop cooperating robots that are able to achieve efficient production planning and scheduling. As of now the sole objective is to create three cooperative robots that work together to transport material over to production machines.

#### 5.2.1.5 RoboCup Junior

This league is an education environment, targeted specifically to youngsters up to the age of 19. Its goal is to provide an environment where learners can expand their knowledge, interest and curiosity about technology at the same time that it stimulates their young and creative minds.

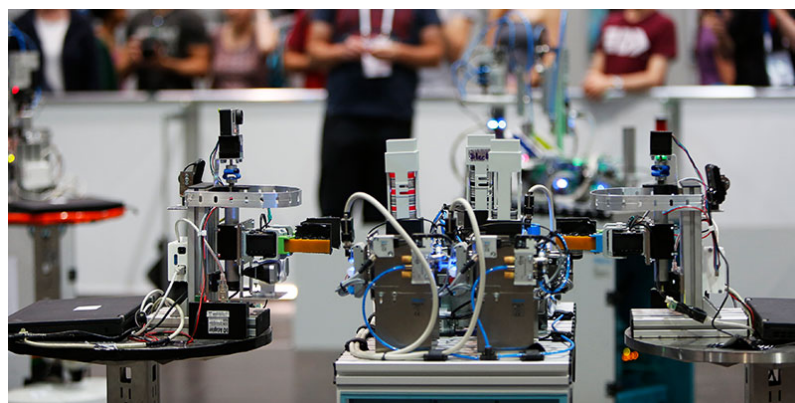


Figure 5.16: RoboCup Industry Logistics (from the official website [2])

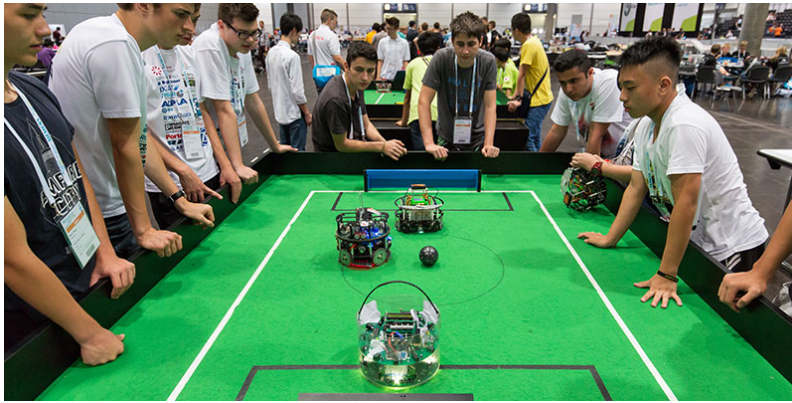


Figure 5.17: Junior Soccer League (from the official website [2])

Using a simple infrastructure, people can create teams of robots to play soccer and rescue victims, much like the normal leagues but with some changes to the rules to make it more compelling and less complicated for the younger future researchers. A challenge that is unique to the Junior Leagues is the *OnStage*, where it invites teams to develop a creative stage performance using autonomous robots that they have designed, built and programmed.

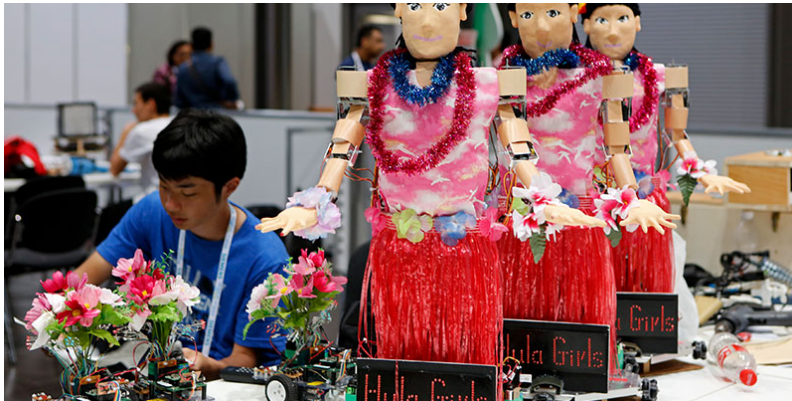


Figure 5.18: Junior OnStage League (from the official website [2])

## 5.3 The RoboCup Soccer Simulator

### 5.3.1 SimSpark

SimSpark is a multi-agent simulator based on the generic components of the Spark 5.3.2 physical multi-agent simulation system and has been used in the *RoboCup Soccer Simulation League* since 2004. It has an established code base with development increasing year-over-year. As the result, RoboCup soccer simulations have changed significantly over the years, going from rather abstract agent representations to more and more realistic humanoid robot games. Thanks to the flexibility of the Spark system, these transitions were achieved with little changes to the simulator's core architecture[57]. Until 2008, the soccer simulator and *SimSpark* simulator were developed and

released as a single project called *rcssserver3d*. It was then broken into two main projects - Spark 5.3.2 simulation platform and *RCSSServer3D* 5.3.3 soccer simulation server to clarify that *SimSpark* is a generic simulation environment, rather than only a robot soccer simulator [57].



Figure 5.19: SimSpark - a multiagent generic simulator

### 5.3.2 Spark

Spark is a physical simulation system. The primary purpose of this system is to provide a generic and flexible simulator for different kinds of simulations. In these simulations, agents can participate in-process or out-of-process.

It has three main components, including the simulation engine, the object and memory management system, and the physics engine.

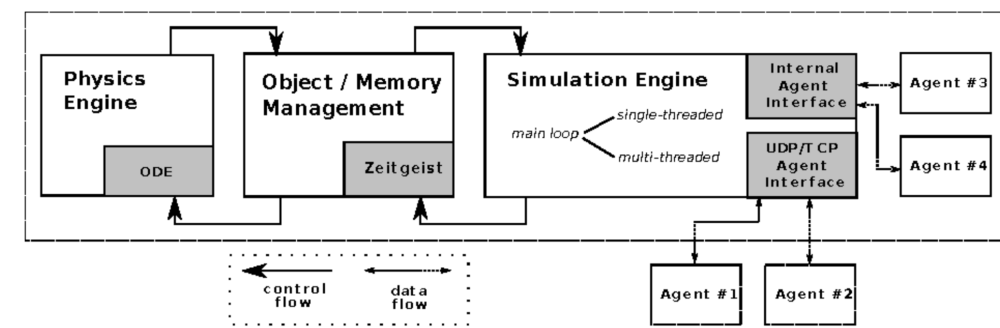


Figure 5.20: Spark Architecture (from [9])

In order to allow soccer games of 11 humanoids players in each team, some changes were implemented, however no modification to the simulator's core was ever specialized for the soccer simulation in order to keep the generalist aspect of the technology.

The changes were [57]:

**Sensor Plugins** - Sensors of a robot allow awareness of the robot's state and the environment.

**Multi-threads supporting** - The physics computation and *SimControlNodes* can run in parallel.



### 5.3.3 SoccerServer

SoccerServer is a simulator of the game of soccer designed as a benchmark for evaluating multi-agent systems and cooperative algorithms [28]. Its architecture is based on client-server, meaning that there are no restrictions to how the teams are built only that they are able to support communications via UDP/IP. It allows the execution of a virtual soccer game played by two teams of 11 virtual autonomous agents (clients), and the possibility of a coach, who are individual processes that connect to a specified port on the server through where all communications are made.

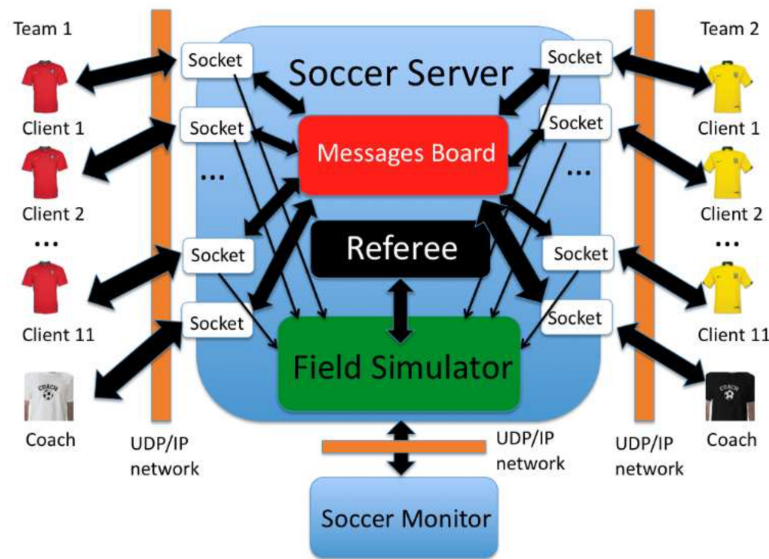


Figure 5.21: SoccerServer Architecture (from [12])

The players send requests to the server regarding the actions they want to perform (e.g. kick the ball, turn, run, etc.). The server receives those messages, handles the requests, and updates the environment accordingly. In addition, the server provides all players with sensory information (e.g. visual data regarding the position of objects on the field, or data about the player's resources like stamina or speed). It is important to mention that the server is a real-time system working with discrete time intervals (or cycles). Each cycle has a specified duration, and actions that need to be executed in a given cycle, must arrive at the server during the right interval [34].

#### 5.3.3.1 Agent Actions on the Server

The agent is limited to a set of actions that are sent by message to the server to execute them [34]. Possible actions are:

- **(turn *Moment*)** The *Moment* is in degrees from -180 to 180. This command will turn the player's body direction *Moment* degrees relative to the current direction.
- **(dash *Power*)** This command accelerates the player in the direction of its body (not direction of the current speed).



- **(kick *Power Direction*)** Accelerates the ball with the given Power in the given Direction. The direction is relative to the the Direction of the body of the player and the power is again between minpower and maxparam.
- **(catch *Direction*)** Goalie special command: Tries to catch the ball in the given Direction relative to its body direction. If the catch is successful the ball will be in the goalie's hand until kicked away.
- **(move *X Y*)** This command can be executed only before kick off and after a goal. It moves the player to the exact position of X (between -54 and 54) and Y (between -32 and 32) in one simulation cycle. This is useful for before kick off arrangements.
- **(turn neck *Angle*)** This command can be sent (and will be executed) each cycle independently, along with other action commands. The neck will rotate with the given Angle relative to previous Angle
- **(say *Message*)** This command broadcasts the Message through the field, and any player near enough, with enough hearing capacity will hear the Message.

#### 5.3.4 RoboViz

RoboViz is an open-source interactive monitor released on February 2011 that renders both agent and world state information in a three-dimensional scene. The main objective of the tool is to facilitate debugging and the analysis of behaviours and algorithms in a sea of data generated on these simulations.

J. Stoecker and U. Visser felt that the existing SimSpark monitor had a number of limitations. In particular, they felt the following issues should be resolved or improved upon [46]:

- **Usability:** the Simspark monitor had a rudimentary interface and the user experience was less polished. For example, the monitor may only be active while the server is online, must be manually restarted with the server, and the window cannot be resized for a higher resolution.
- **Interactivity:** the Simspark network protocol exposes functionality for modifying the game state and moving the players or ball; however, the monitor does not yet make use of these features.
- **Portability:** the monitor is deeply integrated with the Simspark framework making it more difficult to configure, compile, and use.
- **Graphics Quality and Performance:** the Simspark monitor exhibited suboptimal resource usage and performance. While less pressing as other issues, the graphics effects also had significant room for improvement.

The earliest prototype for RoboViz was a program detached entirely from the Simspark framework, but there were serious drawbacks to this design approach that appeared during early prototyping. Unless the agent architecture has access to the world model of the simulation, there is no way to visualize both believed and actual world models simultaneously. Viewing the separate models side-by-side is ineffective in situations where there are discrepancies between what an agent believes and the truth. Furthermore, such a design requires much more effort on the part of a team hoping to utilize the visualization features with this interface [46].

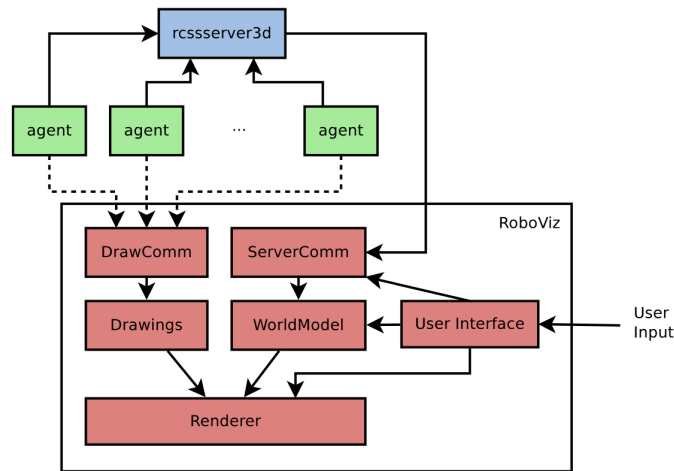


Figure 5.22: Architecture for RoboViz, SimSpark server (rcssserver3d), agent, and user interaction (from [46])

It was concluded that RoboViz would still need to communicate with agents to access their internal states, but it must also use the simulation server's scene graph to render the actual world model. This approach 5.22 provided an opportunity to address many of the SimSpark monitor's deficiencies.



Figure 5.23: RoboCup Soccer Simulation 3D Match in RoboViz

### 5.3.5 Log Player

The Log Player is a feature that has been implemented in most of RoboCup's monitors servers over the years. As the name indicates, its main purpose is to be able to play from a log file. A log file is created during a match and is written to whenever pertinent information is exchanged between the agents and the server that is necessary to recreate a replay of the match in its full-extent. This allows for the teams to have a way of analyzing a past game played, either by mining the file's contents or by replaying it in a supporting monitor that then reads and mimics the players' movements and actions recreating a game of the past.

## 5.4 Conclusions

The AI field is in constant evolution and, with it, so is the robotics fields. A key component in all of this are the autonomous agents that have to either work as a team (cooperative) or against each other (competitive). In order to achieve this they first must be able to communicate, in a structured way, and to act accordingly to achieve their goals. The RoboCup initiative is a great way to improve both these fields and more specifically the research in agent coordination and strategy. In the next chapter, a more in depth approach on the used technologies in soccer simulation are made, along with an overview of relevant literature to the development of this dissertation's project.

Next, a brief introduction to the field of cinematography is made where we go over some of the basic concepts and approaches that will aid in the development of this work.



## Chapter 6

# Project - LiveDirector for RoboViz

The LiveDirector system is a project developed on top of the existing RoboViz with the intent for later approval by means of a pull request to the forked repository, becoming a part of the main visualizer used in the 3D Simulation competitions of RoboCup. The RoboCup international tournament has a large attendance annually with main focus on the Soccer Leagues, of which the 3D Soccer Simulation is part, and for the last couple of years it has been using the RoboViz visualizer to simulate the game played to its spectators, but it lacked the ability to showcase a more real-life soccer viewing experience with a live broadcast feel.

The main objective of the project is to both be able to gather real-time data from the simulation to generate data and parse relevant statistics, similar to those shown in real-life games, and also to provide a live broadcast feel through automatic control and manipulation of cameras positions and angles based on real-time game events.

### 6.1 Architecture

Since the main objective was to implement a real-time live broadcast feel to the visualizer, all development was built on top of the RoboViz application, forked from GitHub. Since both the processes of statistics generation and camera choice rely on the same real-time game events (with exceptions or discarded events in each module), the approach taken was that of developing a single module that would be able to both collect and parse the data information abstracting that process to all the modules that need to obtain that knowledge. Those interested, can subscribe to the module to be notified of when relevant events occur. This is a straightforward example of the *Observer* design pattern well known in software engineering, where a **subject** maintains a list of its **observers** which it notifies whenever there is a state change. A visual representation of the implementation can be found in figure 6.1.

In the diagram we can see all the components of the developed solutions, which we will see in more detail in the remaining sections of this chapter:

1. **StatisticsParser Class** - Responsible for collecting and parsing the real-time data extractor from the match.

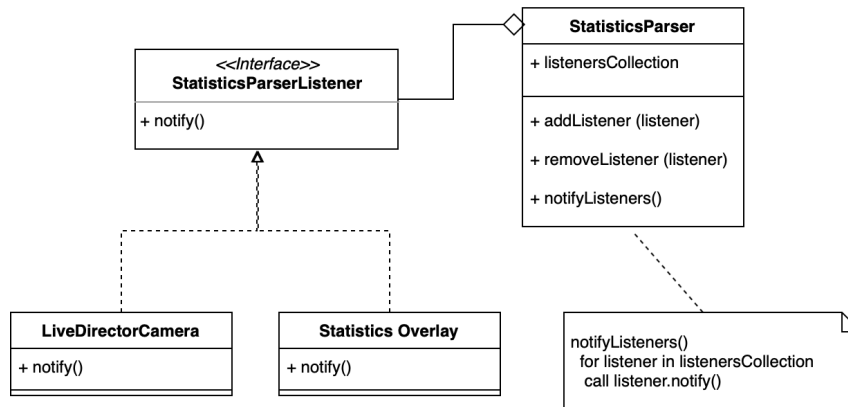


Figure 6.1: Observer pattern implementation

2. **StatisticsParserListener Interface** - The interface that connects the **subject** to its *observers'* methods.
3. **StatisticsOverlay Class** - Organizes screen overlays renderization based on the visualizer' configurations and events received.
4. **LiveDirectorCamera Class** - Manipulates the game camera according to the current game state.

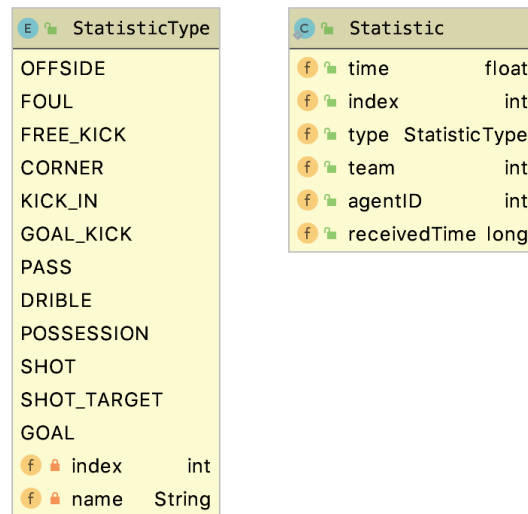
Apart from this architecture, there are two other components implemented that play a more passive, but important role, and that are not my own work, the **Configuration File System** and the **BallEstimator Class** that will be presented further in the following sections.

This implementation allowed for a decoupled development on top of the existing RoboViz application without significant changes to the core software making it suitable for an easy integration into the current main tool. Also, with the implementation through the *Observer* design pattern, new additions to observable events are easy to achieve by simply implementing the event detection algorithm in the *StatisticsParser* class and its respective implementation in the observer classes, through the available *interface*.

In order to share the data between the components in an efficient and organized way, an **object-oriented** approach was used. This simplifies the access to information in a structured and well defined way between all the classes that directly benefit from the direct access to statistics' information. With that in mind, a *Statistic* class was created as shown in figure 6.2. A *StatisticType enum* was also defined in order to keep the statistics' type organized and easily accessible to be compared, if needed, and added to the *Statistic* class as a property.

### 6.1.1 Configuration File System

A configuration file system was already presented in the main version of the RoboViz project and it was directly implemented in this project to take full advantage of its capabilities. An existing *Configuration* class is initialized with a default path and it reads a text file named "*config.txt*".

Figure 6.2: *Statistic* class and *StatisticType* enumerable definition

The file is read line by line and the found configuration variables are used to replace the default values initialized by each configuration structure. The configuration structure for the addition of this project was called *Live Directing Variables* and it is where all configurable variables referenced in the following sections can be set. An excerpt of the configuration file can be seen in listing 6.1.

```

1  ...
2  Overlay Default Visibility:
3  Server Speed           : true
4  ...
5  Statistics Overlay     : true
6  Player IDs             : true
7  Live Directing Variables:
8  Possession Interval    : 5
9  Positions Interval     : 5
10 ...
11 Heat Map               : true
12 Panel Screen Time      : 6
13 Networking Settings:
14 Auto-Connect           : true
15 ...
  
```

Listing 6.1: Configuration File Example

### 6.1.2 BallEstimator Class

The *BallEstimator* class found in this project is a migration of an existing tool from the code repository of the *FCPortugal3D* team, which was firstly introduced to me by Prof. Nuno Lau and

then implemented in Java within RoboViz, from its original C++ source and with all the necessary adjustments.

The goal of the *BallEstimator* is, as the name indicates, to estimate certain aspects about the ball behaviour into future time like its final position, taking into consideration the present visible information about the object, like its consecutive positions and, therefore, its speed. The forces that the ball is subject to are demonstrated in figure 6.3.

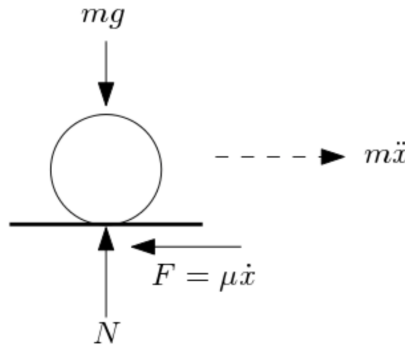


Figure 6.3: Diagram of the forces acting on the moving ball

From here, the functions for its speed [6.1] and positions [6.3] in time,  $t$ , are defined in relation to a constant  $K$  that will aid in the validation of the model.

$$v_t = v_0 \cdot e^{K \cdot (t-t_0)} \quad (6.1)$$

$$x_t = x_0 + \frac{v_0 \cdot (K \cdot (t-t_0))}{K} - \frac{v_0}{K} \quad (6.2)$$

The process of model validation concluded the following value for  $K$ :

$$K = -1.05719 \quad (6.3)$$

And the results of the validation (6.4) show an almost perfect overlap between the real ball positions in future time and the estimated positions by the model.

To go further with the ball estimator, it was made possible to also estimate velocities of the ball in time by fitting observed positions and times on the generated model (see 6.4).

$$estv_2 = \frac{-K}{(e^{-K(t_2-t_1)} - 1)} \cdot (x_2 - x_1) \quad (6.4)$$

Internally, the estimator uses pair of queues of up to 6 values that represent the most recent information regarding the ball whereabouts in order, its positions and time of each. To keep the information updated, the migrated version calls an *update()* method, whenever server time is updated, where these queues are updated. A representation of this class structure can be seen in



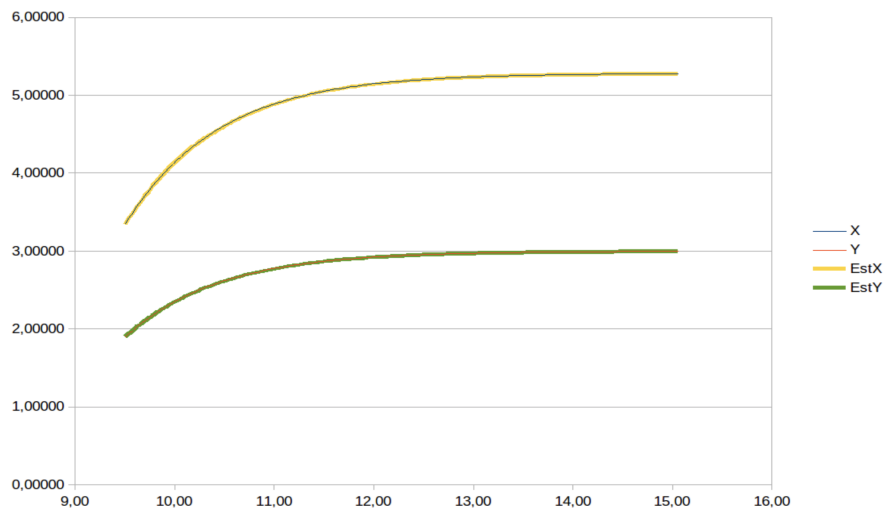


Figure 6.4: Validation of the model results for position estimation

figure 6.5. All its methods are public and called when necessary to estimate either the ball velocity or position at any point in time in the future. You can note that there are also methods that take in no parameters, the *estimatedFinalPos()* functions that estimate the final position of the ball once fully stopped and, therefore, need no reference time input into the future.

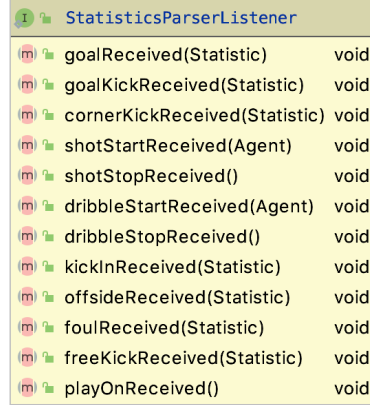
BallEstimator		
m	update()	void
m	estimatedVel(float)	Vec3f
m	estimatedVel4(float)	Vec3f
m	estimatedVel5(float)	Vec3f
m	estimatedVelAway(float)	Vec3f
m	estimatedPos(float)	Vec3f
m	estimatedPos4(float)	Vec3f
m	estimatedPos5(float)	Vec3f
m	estimatedTime(float)	float
m	estimatedTime4(float)	float
m	estimatedTime5(float)	float
m	estimatedFinalPos()	Vec3f
m	estimatedFinalPos4()	Vec3f
m	estimatedFinalPos5()	Vec3f
m	estimatedFinalPosAway()	Vec3f

Figure 6.5: Representation of the migrated BallEstimator Class

### 6.1.3 StatisticsParserListener Interface

The *StatisticsParserListener* interface declares the methods that interested **listeners** must implement in order to act accordingly for each event triggered (see figure 6.6). Different events have

different types of needs and stages so, for a given type, there may actually be more than one trigger associated with it. For example, a *dribble* event has two triggers related to it, one dispatched at the start of a dribble detection and one after the dribble has been given as completed or cancelled.



StatisticsParserListener		
goalReceived(Statistic)		void
goalKickReceived(Statistic)		void
cornerKickReceived(Statistic)		void
shotStartReceived(Agent)		void
shotStopReceived()		void
dribbleStartReceived(Agent)		void
dribbleStopReceived()		void
kickInReceived(Statistic)		void
offsideReceived(Statistic)		void
foulReceived(Statistic)		void
freeKickReceived(Statistic)		void
playOnReceived()		void

Figure 6.6: Representation of the *StatisticsParserListener* Interface

This interface is a major part of the whole architectural approach, filling a role of an intermediate of events between the **subject** and its **observers** without enforcing any own logic.

#### 6.1.4 StatisticsParser Class

As stated in its brief description, the *StatisticsParser* class is responsible for parsing the live game raw data and transform it to specific characterized events that its registered **observers** can listen to and act upon. In order to achieve this we must first analyze how a worthy event can be detected and to what type of statistic it relates to.

##### 6.1.4.1 Statistics Definition

The majority of soccer events, with the exception of those related to game breaks, have similarities. At the origin of this kind of events is always an increase in of the ball velocity or a change in the direction of the ball's motions (named a *kick*), which can represent various events, like a pass, a shot or a dribble. Equation 6.5 shows this concept, where  $t_1$  and  $t_0$  and instants of time and  $V_{ball}$ ,  $D_{ball}$  are ball velocity and direction, respectively.

$$kick(t_0) \leftarrow (||V_{ball}(t_0)|| < ||V_{ball}(t_1)|| \vee D_{ball}(t_0) \neq D_{ball}(t_1)) \wedge t_1 = t_0 + 1 \quad (6.5)$$

The detection of this event serves as an entry point to the detection of all kick-related events such as dribbles, shots and passes which, when detected are converted to game statistics and added to the run-time memory of the parser.

Regarding what will be referred to as *game breaks*, these relate to static events such as free kicks, off-sides, corners and other play modes present in the simulator. Those events are directly managed by the *rcssserver3d* simulation which are then communicated to RoboViz in a structured form and used directly, without much conversion of the data.

#### 6.1.4.2 Statistics Collection

This section dives deeper into the process of data collection and data processing by the *StatisticParser* class for each of the event types defined above, **direct** or **calculated**, respectively. However, to collect information, it is first necessary to have somewhere to store it and, since it will be accessed by many other parties, to easily retrieve it by type.

The most efficient implementation of such a behaviour is normally achieved through a *Map* (or *HashMap*) data structure. In the implementation, the defining pair of the *Map* was  $\langle \text{String}, \text{List}\langle \text{Statistic} \rangle \rangle$ , where the *key* of each of the *Map*'s entries is a string that reflects the fixed name of the *StatisticType* class for the respective type and the value is the list of the statistics collected so far of that given type. By following this approach, we can take advantage of several characteristics that make it a perfect fit for this task, such as:

1. **Fast Lookups** - Lookups on a *Map* take on average  $O(1)$  time, making it one of the fastest, enabling an optimal performance for statistic's lookup even with large amounts of data for a specific type.
2. **Unique Keys** - Since a *Map* does not allow for duplicate keys, we can maintain an organized list of all game occurrences for a given statistic type without the problem of possibly duplicating information or accessing and updating an outdated entry for that type.
3. **Flexible Keys** - Most data types can be used for keys, as long as they're hashable, such as a string, in this case.

The *Map* starts empty without any entries and, as the game progresses, new entries are created for statistics of a type which has no key yet, or existing entries are replaced with their respective updated lists. An example of a *Map* representation at an arbitrary point in time during a match can be seen in figure 6.7.

StatisticsParser.statistics	
"dribble"	[ Statistic1 ]
"shot"	[ Statistic2, Statistic4 ]
...	[ ... ]

Figure 6.7: Representation of the statistics *Map* during a match

### 6.1.4.2.1 Direct Statistics - Game Breaks

As mentioned before, *game breaks* are detected by direct messages sent from the simulation server in a structured way. The structure is in the form of *atoms*, where each atom that comprises it contains a different piece of information about a given message and they can be of variable number between message types, see figure 6.8.

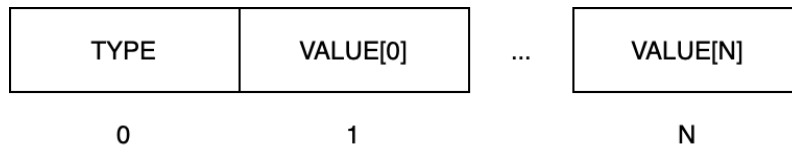


Figure 6.8: Message atomic structure representation

The *StatisticsParser* class is aware of the reception of such messages since itself is an implementer of the *GameState.ServerMessageReceivedListener* interface, exposed by the RoboViz communication's layer, see figure 6.9, which notifies its subscribers at every message received from the server. Those messages exhibit the same structured content that is then parsed in order to extract the relevant information before being inserted into a new instance of a *Statistic* and saved on the run-time *Map*.

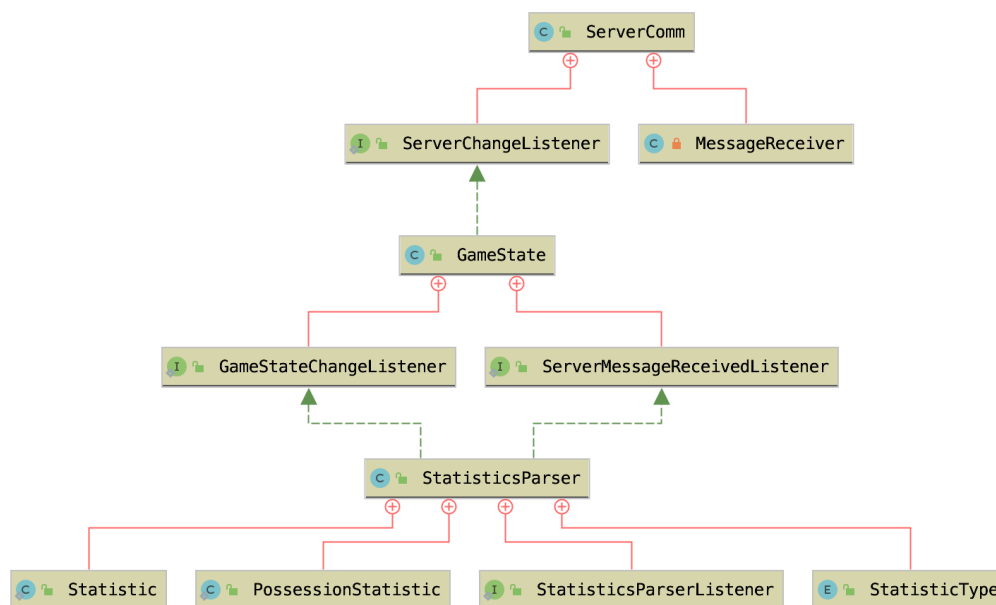


Figure 6.9: UML representing the *StatisticsParser* class hierarchy

The information of the messages' atoms are divided as follows:

1. **Type** - This is the first atom on any message and defines the type of message that is being communicated to the application. It has over 24 possible values but, for the purpose of the *StatisticsParser* class, the relevant values are: "time", "play\_mode" and "foul".
2. **Values** - Each message type can have zero or more value atoms, depending on the amount of information that a message type requires to be fully defined. Each value atom is much like a function parameter, where each ordered value corresponds to an expected information about an event. For each of the relevant message types listed on the aforementioned item, the expected atom values are as follows:

#### 6.1.4.2.2 "time" Message Type

The *time* message receives only a single atom, containing the string value of the actual time of the match in seconds. For example, "254" gets translated to 4:14 minutes of game time. The raw time value is stored on the class at every update to be used during the parsing and calculation of statistics in order to have a real-time setting at each point in the computation.

#### 6.1.4.2.3 "play\_mode" Message Type

For the *play\_mode* message, a single value atom is used (atom of index 1 of the message) to identify the current play mode on the server. Among many values, the relevant one are "KickIn", "corner\_kick", "goal\_kick", "offside", "free\_kick" and "direct\_free\_kick". To each, the strings "\_left" or "\_right" are appended in order to identify to which team the mentioned game event concerns to so.

For example, if a player of the left team is caught offside, the *StatisticsParser* class will receive the structured message depicted in figure 6.10, where the first atom identifies the correct type, "play\_mode" and the second atom (the first value atom) describes the new play mode on the server.

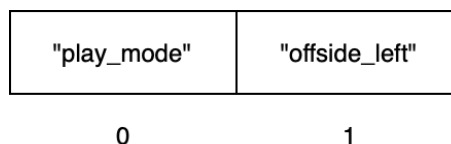


Figure 6.10: Representation of the structured message for an offside for the left team

From here, the *StatisticsParser* will instantiate a new *Statistic* object with the proper converted values of the received message, namely the *type* - "offside" - and the *team* - 1 (refers to the left team, 2 is the right team).

At this point, the run-time class's *Map* would be updated and have a modified entry at the "offside" key, with the newly created statistic appended to its current list of values (see figure 6.11).

StatisticsParser.statistics	
...	...
"offside"	[ ..., NewStatisticN ]
...	...

Figure 6.11: Representation of the class's *Map* after addition of the offside message

Then, the respective *StatisticParserListener* Interface's method, if existent, is called in order to notify the parser's listeners that a relevant *play\_mode* change as occurred. In this case, an *offside*, it would trigger the *offsideReceived(statistic)* method call with the newly created *Statistic* object as a parameter.

#### 6.1.4.2.4 "foul" Message

Of the three, the "foul" message type is the most complete in terms of number of values passed through the value atoms, for a total of 5. A foul message does not follow the same implementation of a play mode message in the sense of describing the culprit's team. There are no appended team identifier strings and instead, the team information is passed through one of the atoms. Along with that data, more follow in the remainder of the value atoms, such as the foul type and agent id. A clear representation of this structure can be found on figure 6.12. The message type identifies the foul message followed by the ordered value atoms that describe the message index, the foul type, the offending team and the specific player, respectively.

"foul"	"1234"	"crowding"	"2"	"9"
0	1	2	3	4

Figure 6.12: Representation of the structured message for a crowding foul by left team's player 9

The conversion of this message type will generate more initialized fields on the new instance of a *Statistic*, namely the *agentId* property, which is not initialized at a play mode message since that information is not transmitted. As an example, the message depicted in figure 6.12 would translate to a *Statistic* object with a *type* of "foul", *index* of "1234" (unique identifier), *team* of value 2 (which is directly converted from the original message), and *agentId* of 9.

As before, the run-time's statistics *Map* is updated and the "foul" entry get modified to reflect the newly parsed statistical information (see figure 6.13).

StatisticsParser.statistics	
"offside"	[ Statistic1, Statistic52 ]
"foul"	[ ..., NewStatisticN ]
...	...

Figure 6.13: Representation of the class's *Map* after addition of the foul message

Same as a message of type "play\_mode", the respective **subject** interface's method is called, *foulReceived(statistic)*, to notify the observers that a foul has taken place so that they can act accordingly.

#### 6.1.4.2.5 Calculated Statistics

Calculated statistics are those that cannot be inferred directly from received messages of the controlling server and, instead, must be detected through run-time algorithms that constantly parse real-time data being received regarding the game's state. There were implemented different types of statistics generation tools that each generate its relevant data, identified as:

1. **Direct Parsing** - As the name implies, direct parsing tools calculate statistics based on directly manipulating data about the game's current state.
2. **Event Driven** - These tools rely on series of occurring events in order to detect the existence of relevant moments in player behaviour.

We will now see which statistics were generated, which type of tool of the aforementioned was used and how the data was collected and stored.

#### 6.1.4.2.6 Possession

Possession is collected through means of **Direct Parsing**. At set intervals (configurable), the parser will run the Possession Detection algorithm see( 5) where it loops through all the players on each team and calculates their weighted possession score according to the following expression:

$$P = \alpha d + \beta t \quad (6.6)$$

The formula calculates each individual player's possession score based on a weight ( $\alpha$ ) of their current distance ( $d$ ) to the ball and a weight ( $\beta$ ) of the interval of time their team last had possession ( $t$ ). The possession is awarded to the team of the player with the lowest calculated score, at which point the interval of team possession is updated for each of the teams.

**Algorithm 5:** Possession Detection

---

```

1 lowestScore  $\leftarrow$  MAX_SCORE
2 winnerPlayer  $\leftarrow$  null
3 winnerTeam  $\leftarrow$  null
4 foreach team in teams do
5   | players  $\leftarrow$  team.getPlayers()
6   | foreach player in players do
7   |   | score  $\leftarrow$  calculateScore(player,team)
8   |   | if score < lowestScore then
9   |   |   | lowestScore  $\leftarrow$  score
10  |   |   | winnerPlayer  $\leftarrow$  player
11  |   |   | winnerTeam  $\leftarrow$  team
   |   | end
   | end
end
12 updateTeamPossessionIntervals(winnerPlayer,team)

```

---

After several tests, the best results were achieved with the following factor values:

$$\alpha = 1, \beta = 0.4 \quad (6.7)$$

In the end, the distance to the ball was given full accountability to the final score and the time interval was given less weight due to its high impact on larger values. This approach was taken in order to account for situations where, for example, a player was dribbling and between the time where he moves the ball forward and reaches it again, an opponent may have been at a closer distance to the ball, but that does not mean his team had possession of it.

Table shown in figure 6.14 presents the results for some given examples of team possession interval time ( $t$ ) and distance of the closest player to the ball ( $d$ ). If a touch by the opposing team is detected, both team's time intervals are reset. This is done due to cases like deflections during a pass, for example. In a deflection we can neither assume that the deflecting team as won the possession back or that it still belongs to their opponent, therefore we reset the time factor of the model and let the teams "battle" for possession once again.

Once the algorithm has run and detected the correct possession, a new *Statistic* object is created with the collected information - *agentId*, *team*, *time* - generating a new statistic of type *POSSESSION* which is then added to the statistics *Map* of the class.

#### 6.1.4.2.7 Heat Map

The generated **center-of-action** heat map also relies on **Direct Parsing** techniques, reading directly from the game's state. Again at configurable set intervals, the ball position is collected



	Distance (m)				
Time (s)	1	2	3	4	5
2	1,8	2,8	3,8	4,8	5,8
5	3	4	5	6	7
10	5	6	7	8	9
15	7	8	9	10	11
20	9	10	11	12	13
25	11	12	13	14	15

Figure 6.14: Possession scores for given values example

and its 3D representation is then converted into a coordinate pair that will represent its position within a matrix, where each cell corresponds to a specific area of the field.

In RoboCup, positions are given relative to the center point of the pitch, where the position along the field's plane corresponds to  $(0, 0)$ . Due to this implementation, each corner would then have negative and positive position coordinates depending only on the field's width ( $W$ ) and length ( $L$ ). An illustrative representation of this can be seen in figure 6.15.

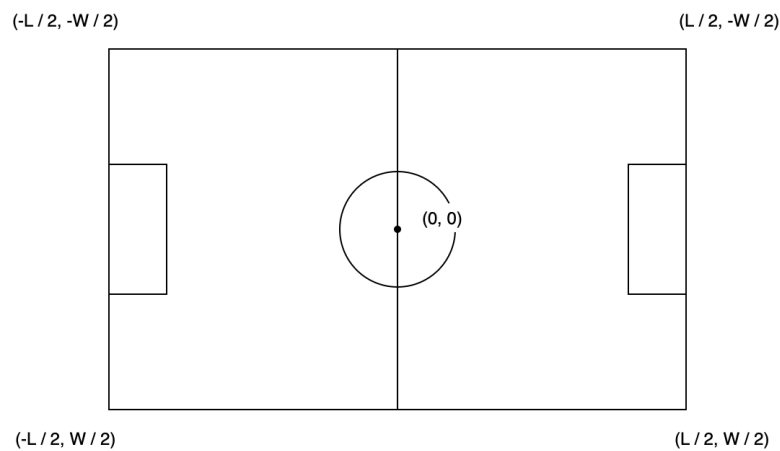


Figure 6.15: Representation of the field's coordinate system

To represent the positions on the field in a structured way, the most intuitive approach is to implement a matrix where each of its cells represents a specific area of the pitch. To represent a matrix within most programming languages, the approach is to declare a bi-dimensional array. In this specific case, the array created would have a first dimension of  $W$  (rows in the matrix) and a second dimension of  $L$  (columns of the matrix), see figure 6.16 for a visual representation. At its initial state, all cells have a value of 0, the ball has been in each represented pitch position 0% of the game's duration.

After defining the storage structure for the data, it needs to be updated. The first step in this



a shot is the magnitude of the ball's velocity change, which can be set on the configuration file. Once this is confirmed, the final step is to determine whether the ball's final position is within the goal attacking area of the attacker's team or if its trajectory as crossed it. This attacking area was separated into two distinct zones, the zone considered to be a shot on target and the zone for when that is not true (see figure 6.17).

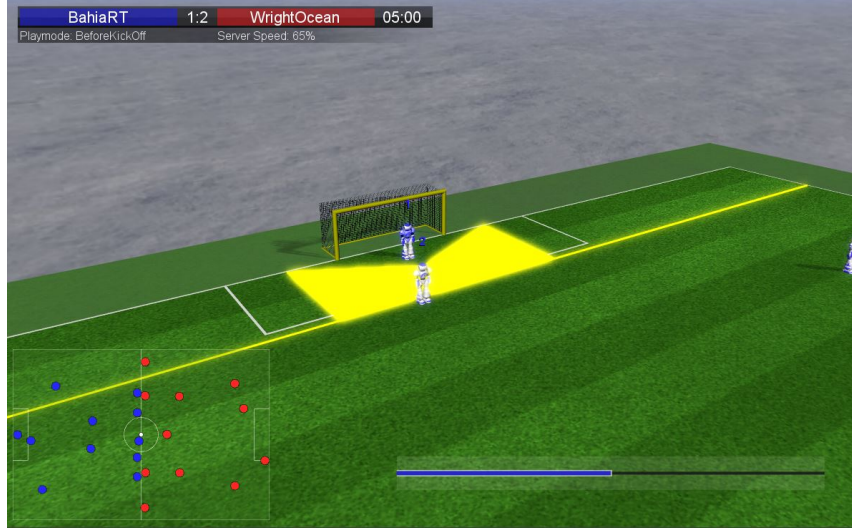


Figure 6.17: Representation of the attacking zones of the right team

The zone that triggers a shot on target is delimited by a gap variable configurable through the file as well, **shotDistanceTrigger**, that sets a goal side distance gap and a goal forward distance gap. Any final position of the ball that is to either side of the maximum goal side distance gap and within the forward distance, is considered a *shot off-target*.

Now, for clear reasons, we cannot simply see if the ball's final position is within each of the zones and attribute the *shot* type directly, because the ball could end up behind the goal line outside of the *on-target* y-delimited area but still have been a *shot on-target* (see figure 6.18).

So we conclude that is necessary to analyse the ball trajectory instead of its final position. A general approach to determine if the ball has **passed through** the *on-target* zone is to check if its trajectory vector intersects any of its delimiting lines on the *x-axis*. A few examples of trajectory vectors are described in figure 6.19 to better illustrate all possible shooting scenarios and that this approach is suitable to all.

In all cases, even if the ball trajectory itself does not, the ball's trajectory vector will always cross at least one of the *x-axis* lines that delimit the area. So the calculations are always done with the trajectory vector and not the ball trajectory it self.

In the end, all that is needed for the calculations are the ball's trajectory vector and zone's delimiting imaginary lines. The former is obtained (with the help of the *BallEstimator* class) through the simple subtraction of the ball's final position to the shot origin (shooter's position):

$$\vec{T}_{ball} = FinalP_{ball} - P_{player} \quad (6.10)$$

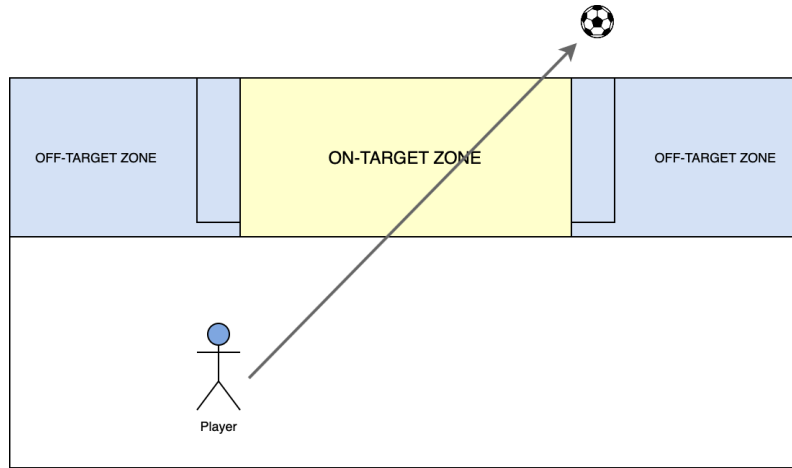


Figure 6.18: Representation of the case of an *shot on-target* that ends outside in the *off-target* y-coordinate space

The  $FinalP_{ball}$  is calculated by the *BallEstimator* class and retrieved through the *estimatedFinalPos* function family described previously, that returns an estimated position for the ball in the future. The latter are obtained through the configuration file variables and their addition to the field's real dimensions.

Once we have this information, we check if the ball's trajectory vector,  $\vec{T}$ , has a point that intersects any of the  $x$ -axis lines that delimit the zone. The y-coordinate of that point,  $I$ , at a given  $LINE\_X$  is calculated by:

$$I_y = Player_y + (\vec{T}_y \cdot (LINE\_X - Player_x) \cdot \vec{T}_x) \quad (6.11)$$

If the intersection's point y coordinate,  $I_y$ , is within the  $[-y, y]$  of the zone, then it is inside the *on-target zone*. If not, we repeat the same process for the goal line's  $x$  ( $L/2$  or  $-L/2$  depending on the attacking team's side) and do the same interval check.

At the end, a shot is considered either a *SHOT* or a *SHOT\_TARGET* and a new *Statistic* is created and assigned the resulting type, involved player and respective team. It is then added to the statistics *Map* under the corresponding key, and a notification is sent to all **observers** with the created object in a call to the *StatisticsParserListener* interface's method *shotStartReceived(player)*.

If at any point during a *shot*, the ball's trajectory was to suffer a major deviation from the originally calculated one (hit another player), the shot would be declared as over and the *shotStopReceived()* method would be called.

#### 6.1.4.2.9 Dribble Detection

Same as the **Shot Detection** discussed above, a dribble detection has the entry point of a *kick* however, the rest of the process is quite simpler. On each *kick* detection that has not been classified

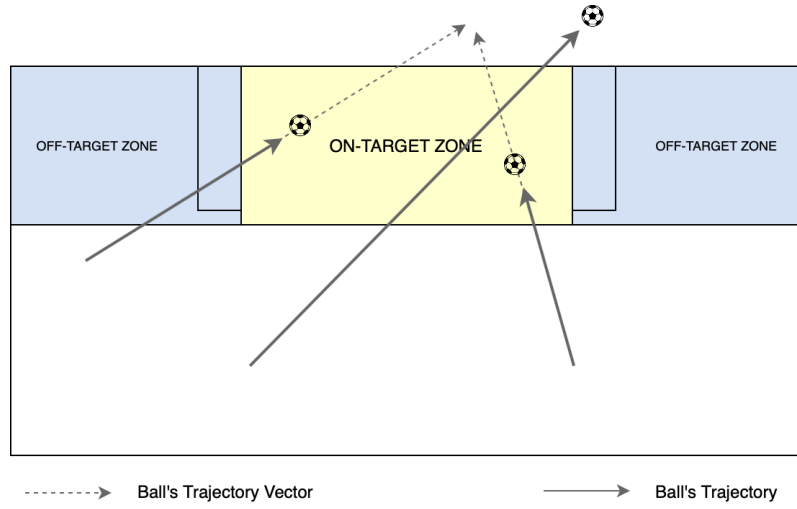


Figure 6.19: Representation of the possible shooting scenarios

as a *shot* the following validation is made:

$$Dribble(t+1) \leftarrow kick(t) \wedge Player(t).id = Player(t+1).id \wedge Player(t).team = Player(t+1).team \quad (6.12)$$

If the above is true, then an internal variable that keeps track of the number of touches is incremented and the *dribbleStartReceived(player)* method is triggered to notify the event start. If it is false, the counter is reset, the reverse method is called, *dribbleStopReceived()* and a dribble is attributed to the previously detected player if the following logic holds truth:

$$Dribble_{player} \leftarrow ballTouches \geq MIN\_DRIBBLE\_TOUCHES \quad \wedge \quad dribbleDistance \geq MIN\_DRIBBLE\_DISTANCE \quad (6.13)$$

Both *MIN\_DRIBBLE\_TOUCHES* and *MIN\_DRIBBLE\_DISTANCE* are configurable constants and correspond to the minimum number of touches a player must have on the ball without losing possession and the minimum distance between the starting and finishing points to be considered a dribble, respectively.

Finally, to record this event, a new *Statistic* object is created with type *DRIBBLE* and the player and team attributes are set. It is then added to the current *Parser's Map* under the "dribble" key, ready to be accessed.

### 6.1.5 StatisticsOverlay Class

The *StatisticsOverlay* class's job is to coordinate the rendering of its own overlay on the RoboViz's UI, when enabled. The default setting is loaded from the configuration where it can be set as *true*

or *false* to be either enabled or disabled, respectively. Additionally, a menu item was added to be controlled via the UI itself with its unique shortcut keybind as well (see figure 6.20).

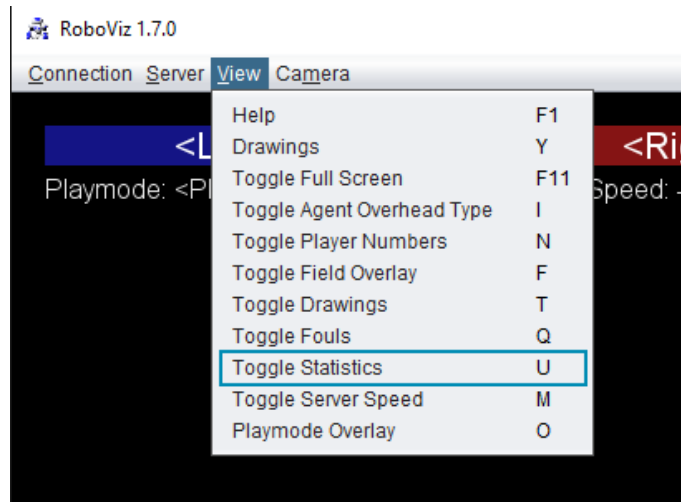


Figure 6.20: Menu item for toggling the Statistics overlay

#### 6.1.5.1 StatisticsPanel

In order to facilitate and organize the structure of commonly displayed information type a *StatisticsPanel* class was created in order to take an object-oriented approach on information manipulation. The class is very basic and its purpose is to only keep an list of **titles** and **values**, that will be rendered on screen (see figure 6.21).

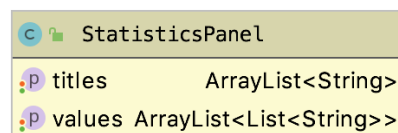


Figure 6.21: Representation of the *StatisticsPanel* class

Instances of this class are dynamically built at run-time whenever panel-type information is to be displayed on the screen. Each *title* added to the list corresponds to a line in the final render of the panel. Respectively, for each *title* entry on list, there is an entry on the values list. This entry is a tuple, where the first value corresponds to the left team and the second value to the right team. A visual representation of the final rendered panel structured content can be seen in figure 6.22.

#### 6.1.5.2 State Management

Since all statistic collection and notification and handled by the *StatisticsParser* class, the *StatisticsOverlay* class is only responsible for managing which graphics to display and for how long.

PanelN		
values[0][0]	titles[0]	values[0][1]
values[...][0]	...	values[...][1]
values[N][0]	titles[N]	values[N][1]

Figure 6.22: Representation of the *StatisticsPanel* class

All panels start off disabled, except for the **heat map** that is always on or off depending on the initial configuration read from the file. For the remaining available panels, all are controlled by the reception of the respective notifications. The exception to this rule are the possession panel and graph that is controlled internally, being triggered every 60 seconds (configurable).

To keep track of whether the panels should be displayed or ignored when the trigger is received, a *Map* is kept storing the last display time of each panel. This panel is initialized with keys relating to all possible receivable events and a value of 0 (see figure 6.23).

panelLastTimes	
"foul"	0
"possession"	0
...	0

Figure 6.23: Representation of the *StatisticsPanel* class's *Map* of panel times

As the game progresses, events are received through the *StatisticsParserListener* interface's methods and, if no current panel is being displayed at the time, the correct panel type is updated according to the event type description - "foul", "dribble", etc - and its *Map* time renewed (see algorithm 6) only if the last render of that panel type happened at least *PANEL\_TIME\_OFF* seconds ago (also configurable). The only exception to this behaviour is, as stated above, the possession statistics that are not triggered by the *StatisticsParser* itself but by the overlay class (since it is not an event but a constant collection).

**Algorithm 6:** Panel update logic

---

**Input:** *panelType*

```

1 if panelVisible then
2   | return;
  end
3 panelLastTime  $\leftarrow$  getMapKey(panelType.name)
4 if CURRENT_TIME - panelLastTime > PANEL_TIME_OFF then
5   | setMapKey(panelType.name, CURRENT_TIME)
6   | currentPanel  $\leftarrow$  panelType
  end

```

---

After a few minutes of gameplay, the timings *Map* will have a few more entries with updated times (see figure 6.24).

panelLastTimes	
"foul"	21,35
"dribble"	42,67
...	...

Figure 6.24: Representation of the *StatisticsPanel* class's *Map* of panel times after a few seconds of gameplay

### 6.1.5.3 Panel Rendering

Once the entirety of the panel state management is processed, the class has all the necessary information to rendered the correct arrangement of panels, always-on or not. Bellow is briefly explained how each statistical gadget is put together and rendered on screen.

#### 6.1.5.3.1 Timeline

The Timeline gadget is a type of always on or off gadget, that is defined through the configuration file at startup. This gadget is the simplest of the collection and at its core it is a simple polygon of ever-increasing width that also renders events at specific points in time (see figure 6.25).



Figure 6.25: Timeline gadget on a running game with goals

With the aid of the *drawBox* method, a fixed background is drawn at a given point (*x,y*) with a specific width calculated as a factor of the total width of the view-port. On top of this, another



rectangle is rendered with a different height and at the middle of its background. It makes use of the same *drawBox* function but, this time, the width parameter passed into is calculated by:

$$Line_{width} = \frac{time}{TOTAL\_GAME\_TIME} \cdot Timeline_{width} \quad (6.14)$$

The same applies to the events that are rendered within the timeline. At this version of this gadget, only goal events are considered and rendered. Being an always-on gadget, it continuously collects the goal statistics from the *StatisticsParser* (usually not very numerous) and, cycling through them, renders each one at the top or bottom of the timeline (depending on the team associated with that statistic) and at the  $x$  calculated with the same equation 6.14.

The symbol used to identify the goal event is a simple *GL\_POINT* of the same color of the team that has scored and a fixed size with a larger point behind it to give the idea of a border.

### 6.1.5.3.2 Heat Map

The Heat Map gadget is also a type of always on or off gadget, that is defined through the configuration file at startup. It makes use of the ball's position matrix that the *StatisticsParser* is constantly updating and manipulates a copy of that information to achieve its individual requirements. The first manipulation is the calculation of the occupation percentage values on each of the matrix's cells by dividing each cell's position count value by the total value of positions recorded (all accessible on the *StatisticsParser* class publicly). The end result is a matrix of the same dimensions with decimal ratios on each cell (see figure 6.26).

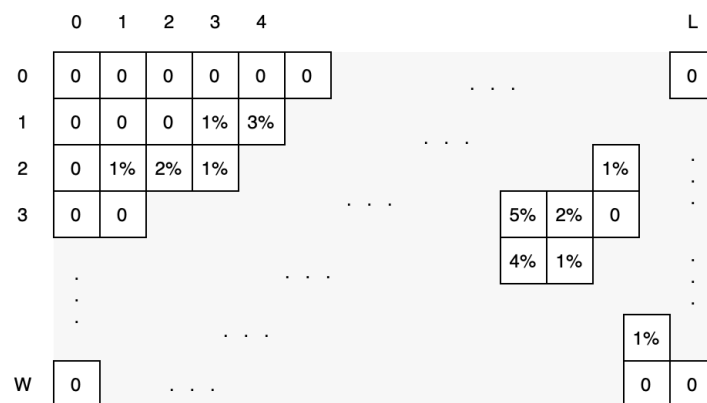


Figure 6.26: Representation of a calculated occupation percentage matrix

After generation the *ratio's matrix*, each cell is rendered on screen in the same order they appear in the matrix, from top to bottom and left to right. This is abstracted through a looped called to a *drawBox3f* method that receives 4 parameters and sets the four vertices of a box defined by them, similar to the examples in section 3 (see 6.2). Those parameters define the bottom left vertex of each cell and its width and height, which are given by dividing the real field dimensions

by the number of cells along each of those axis (the rendering happens within a projection view of the same dimensions of the field that is then placed on the screen resized).

```

1  static void drawBox3f(GL2 gl, float x, float z, float w, float h)
2  {
3      gl.glVertex3f(x, 1, z);
4      gl.glVertex3f(x + w, 1, z);
5      gl.glVertex3f(x + w, 1, z + h);
6      gl.glVertex3f(x, 1, z + h);
7  }

```

Listing 6.2: Draw box method

Finally, in order to output a consistent coloring to simulate a normal heat map behaviour (a stronger color indicates more usage and a fainter one indicates low usage), a normalization of each ratio was done in relation to the highest ratio recorded (see 6.15). This way, the highest calculated ratio in the matrix will have the strongest color (full opacity,  $\alpha = 1$ ) and consequent values will have a non-linear opacity variation based on the highest up to the lowest value that will be always capped at a certain visible opacity level, *DESIRED\_MIN*.

$$\alpha = (1 - DESIRED\_MIN) / (MAX\_RATIO - MIN\_RATIO) \cdot ratio + 1 - ((1 - DESIRED\_MIN) / (MAX\_RATIO - MIN\_RATIO)) * MAX\_RATIO \quad (6.15)$$

The final result can be seen in figure 6.27. In this example, we can conclude that the right team had a more offensive game with larger presence of the ball in their opponent's side. Also, the peak of the game's dispute was around the mid-field circle.

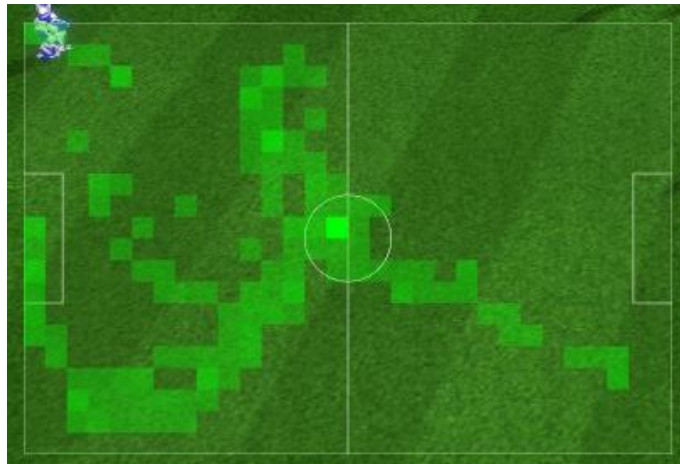


Figure 6.27: Heat Map output at the final stages of a match

### 6.1.5.3.3 Statistical Panel

Every time a panel type is defined on the class (not *null*) a *StatisticsPanel* object is updated with the respective information. In order to keep this approach extensible, each panel type can show any collection of statistics gathered from the *StatisticsParser* collection. For example, in the event of a corner, the "*Corner*" title is added to the panel object, as well as the corner count for each team (see 7).

---

**Algorithm 7:** Panel build for a *corner* event reception

---

```

1  panel ← createPanel()
2  switch panelType do
3      ...
4      case CORNER do
5          cornerStatistics ← StatisticsParser.get("corner")
6          panel.titles[0] ← "corner"
7          panel.values[0]
            ← [getLeftTeamCount(cornerStatistics), getRightTeamCount(cornerStatistics)]
            end
8      ...
    end
end

```

---

If in the future someone was to add new calculated or direct statistics that directly related to corners and wanted to show them at a corner event, the specific panel build could be changed to reflect their needs. An hypothetical example can be seen in figure 8 where besides the corner counts for each team we also added the foul count. The same is true for any type of received event and for any statistical information that can be retrieved from the *StatisticsParser* instance.

**Algorithm 8:** Panel build for an hypothetical new *corner* event reception

---

```

1 panel ← createPanel()
2 switch panelType do
3   ...
4   case CORNER do
5     cornerStatistics ← StatisticsParser.get("corner")
6     panel.titles[0] ← "corner"
7     panel.values[0]
8       ← [getLeftTeamCount(cornerStatistics), getRightTeamCount(cornerStatistics)]
9     foulStatistics ← StatisticsParser.get("foul")
10    panel.titles[1] ← "foul"
11    panel.values[1]
12      ← [getLeftTeamCount(foulStatistics), getRightTeamCount(foulStatistics)]
13  end
14 end

```

---

With the panel object correctly instantiated with all necessary information, its rendering is quite simple. Firstly, the team names are rendered on top, to be able to tell which values correspond to which team. This is done by modifying an existing method called *drawTeamNames* that was used to display the team names bar along with other information like goal count and game time. All the unnecessary information was removed and only the team's names remained.

To render the statistical information, the first step is to obtain the number of lines to be rendered, which corresponds to the length of the *titles* property of the panel object. This allows us to dynamically render a box of enough height to hold all the values inside enabling for any custom case of event statistics' count. This box is rendered from a fixed point, (*x*, *y*) that sets the top left corner of it. From there, three more vertices are set with the set *width* of the box and the calculated height (see 6.3).

```

1  int numberOfLines = panel.getTitles().size();
2
3  gl.glBegin(GL2.GL_QUADS);
4  gl.glColor4fv(panelColor, 0);
5  gl.glVertex2fv(new float[] {x, y}, 0);
6  gl.glVertex2fv(new float[] {x + w, y}, 0);
7  gl.glVertex2fv(new float[] {x + w, y - (numberOfLines + 0.5f) * LINE_HEIGHT},
8  0);
9  gl.glVertex2fv(new float[] {x, y - (numberOfLines + 0.5f) * LINE_HEIGHT}, 0);

```

Listing 6.3: Draw dynamic panel box

Then, on top of the rendered background box, the text is written in each line by decreasing the *y* coordinate of the text renderer by the same *LINE\_HEIGHT* used to generate the background dimensions. To keep the text centered within each title cell and not aligned left, the calculation

for each cell's center point is done and then the text renderer pointer is set back exactly half the width of the text so that, when it is finally rendered, it will be perfectly centered. This is a common approach to center text and the implementation can be seen in 6.4.

```

1   for (int i = 0; i < panel.getTitles().size(); i++) {
2       tr2.draw(panel.getTitles().get(i), x + w / 2 - (int) tr2.getBounds(panel.
        getTitles().get(i)).getWidth() / 2,
3           LINE_Y);
4       tr2.draw(panel.getValues().get(i).get(0), x + Y_PAD, LINE_Y);
5       tr2.draw(panel.getValues().get(i).get(1),
6           x + w - Y_PAD - (int) tr2.getBounds(panel.getValues().get(i).get(1)).
        getWidth(), LINE_Y);
7       LINE_Y -= (LINE_HEIGHT);
8   }

```

Listing 6.4: Render the statistical information text centered

At the end of the process, the generated panel can be seen has in figure 6.28.



Figure 6.28: Foul panel at a late stage of a match

#### 6.1.5.3.4 Possession Graph

The possession graph, contrary to the heat map, is not an always-on gadget but, as the heat map, it is configurable to be displayed or not in the configuration file. It is displayed only when the possession panel is also shown on screen and it is enabled by configuration. The goal of this graph is to depict the possession variation of each team through the match. The approach taken was of a disjunctive area chart, since the possession values of each team are ways the complementary of the other. In figure 6.29 we can see the concept in action. As the game starts, the team that handled the kick-off (blue team, in this case) has a possession of 100% at the start then, as the game progresses we can visually understand how the possession evolved over time for each of the teams. The white line at the middle of the graph indicates exactly that, the middle point of possession, the 50% mark.

The idea behind this gadget was fairly simple but its implementation was trickier. In the end, since it is a disjunctive graph, we only need information about one of the sides, since the other since in the complementary value of the former so it was arbitrarily chosen to always construct the graph based on the left team.

This gadget has in itself two backgrounds, the main background in which the title and the graph are inserted into, and the actual graph background. As before, the backgrounds are rendered



Figure 6.29: Possession graph at a late stage of a match

with the aid of the *drawBox* method that sets a rectangle's vertices based on the starting point and a given width and height. After the outer box is rendered, an inner box follows. In this case, this inner box is rendered with the color of the right team and filled in its entirety so, the graph background itself is a rectangle of the right team's color. At this stage, if we were to render the gadget on screen, it would be nothing more than two colored rectangles (see figure 6.30).

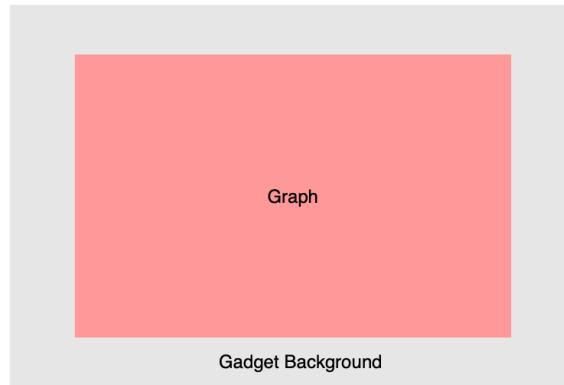


Figure 6.30: Possession graph at render stage 1

At this stage, what is left now is the rendering of the left team possession stats over time. This is done by looping in pairs the possession statistics that are stored in the *StatisticsParser* class. For each record in this collected list, a colored quad is rendered. Its vertices are a connection between itself and the previously rendered quad.

Both bottom vertices of the quad always have a set height ( $y$ ) of 0 relative to the graph window, being flush with its base. Regarding their  $x$ , it is the result of the addition of the graph's base left  $x$  coordinate with the statistic time converted to a scale of the graph's width (see 6.16).

$$Quad_x = Graph_x + \frac{statistic_{time}}{TOTAL\_GAME\_TIME} \cdot Graph_{width} \quad (6.16)$$

The calculation of its height follows exactly the same approach but this time it is a factor of the possession value on the graph's height scale (see 6.17).

$$Quad_y = Graph_y + statistic_{possession} \cdot Graph_{height} \quad (6.17)$$

With this, four vertices are set with the left pair being the right pair of the previous quad rendered, and the right pair being the results of the above calculations (see figure 6.31). The result is a list of polygons that together form a complete area chart.

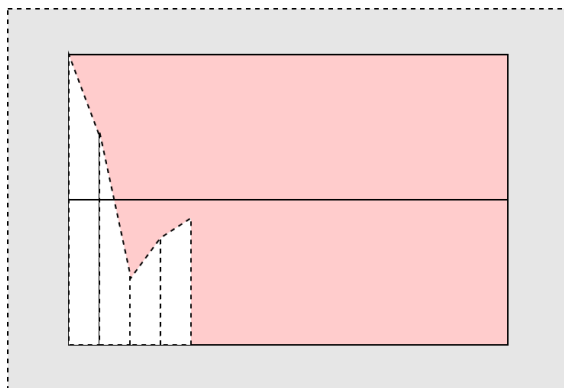


Figure 6.31: Possession graph polygon calculation illustration

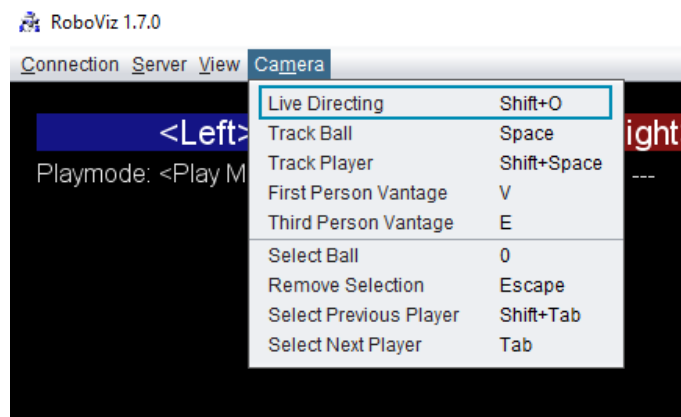
### 6.1.6 LiveDirectorCamera Class

The *LiveDirectorCamera* class's goal is to manage the Roboviz's camera manipulation, when enabled. The default setting of the RoboViz application is a fixed camera that points to the center of the pitch and can be controlled by a user with the mouse and arrows keys. There is also another type of camera that can be toggle through the *Spacebar* or an item on the *View* menu and that follows the ball at a fixed angled around the pitch. To activate the new camera director developed through this work, a menu item was added to be controlled via the UI itself with its unique shortcut keybind as well (see figure 6.32).

#### 6.1.6.1 State Management

As with the other camera controllers available at RoboViz, the first step to activate a given controller is to set it to *enabled* (and disable the current active one). Once activated, all state is controlled by itself and the reception of the triggered events sent through the *StatisticsParserListener* interface.

At initialization, a list of camera positions is set and, as of this initial version, their virtual position is layed out as seen in figure 6.33. As you can see, there are only two named camera types - CAM\_1 and CAM\_2 - but three overall cameras on the pitch. That is why they were referred to as virtual cameras since, internally, there is a list of configurations that define dynamic camera positions based on team side, not duplicating the configuration but simply altering its values. In the

Figure 6.32: Menu item for toggling the *LiveDirector*

end, most of the work is done by the CAM\_1 and the remaining, secondary, cameras are triggered at set pieces, in this case at the goal kick play mode.

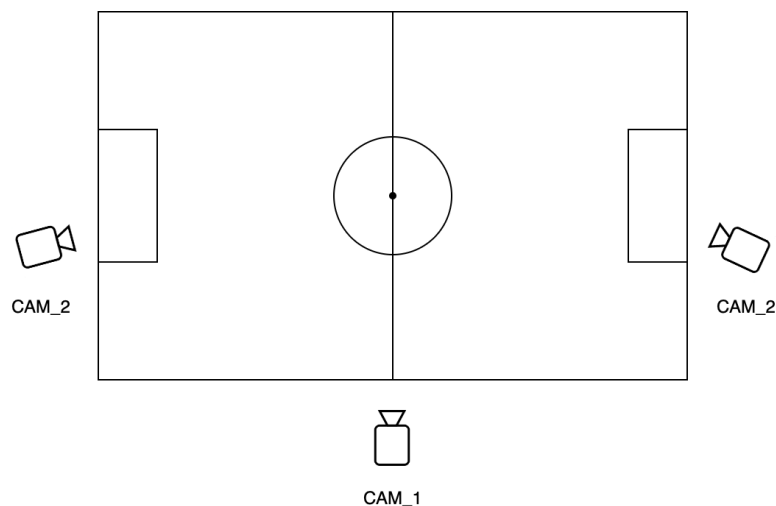


Figure 6.33: Virtual positions of the set cameras

At the reception of an event, the respective method is called on the *LiveDirectorCamera*'s side and specific actions are taken. As the *StatisticsParser* class, there are essentially two types of state updates that alter the behaviour of the *LiveDirectorCamera* class, the direct and the calculated events. **Direct** events in this camera module are responsible only for updating the current camera configuration, meaning that they change the current camera view between the previously set configuration values. These are used to switch into the set pieces cameras like the goal kick mentioned before, and to revert back to the main camera feed once the play mode has been reset. **Calculated** events influence the live behaviour of the main camera by altering its intended filming target causing several other adjustments.

Once the correct camera type has been set for the current situation, the class's *update()* function, called constantly by its parent, RoboViz's *Viewer* class, will handle the respective update behaviour expected for said type.



As for this version of the work, bellow will only be explained the control of the *LIVE* feed camera (CAM\_1) since the set pieces are static cameras that, once set, do not possess much dynamic control.

### 6.1.6.2 Camera Control

The *LIVE* camera feed works in its essence, as a target tracker meaning that, given a target, it controls it self to keep that target on sight, according to some possible preferences. The levels of control manipulated in order to achieve the desired results are the camera's pan, tilt and zoom. A combination of these three aspects gives a real life feel to the resulting broadcast.

#### 6.1.6.2.1 Target Tracking

The *LiveDirectorCamera* maintains an internal variable where it stores its *target*, which is initially set to be the ball object. Once the target is set, the update cycle for the *LIVE* starts.

The camera's pan is a direct interpretation of the target's position. Given that the camera is fixed at the center line, the pan angle,  $\theta$ , is the angle between the center line across the field and the imaginary line at the field's plane that connects the camera to its target (see figure 6.34).

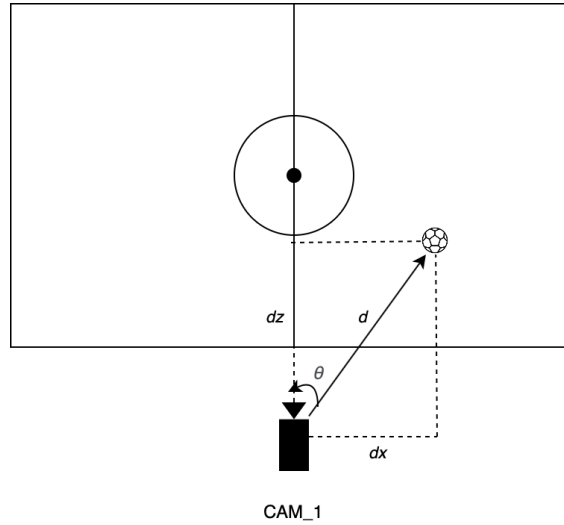


Figure 6.34: Diagram of the pan angle calculation

Since we can directly calculate all the necessary distances between the objects, the angle is obtain through the simplest trigonometric principles (6.18), with the distance between camera and target,  $d$ , being the direct length of the difference of their positional vectors.

$$\theta_{pan} = 180 + \text{asin}\left(\frac{\text{target}_x - \text{camera}_x}{d}\right) \quad (6.18)$$

The same is true for the camera's tilt. The tilt angle can also be achieve through the application of the same trigonometric principles as for the pan (see figure 6.35). However, a fraction of the resulting angle is applied to dampen the tilt change.

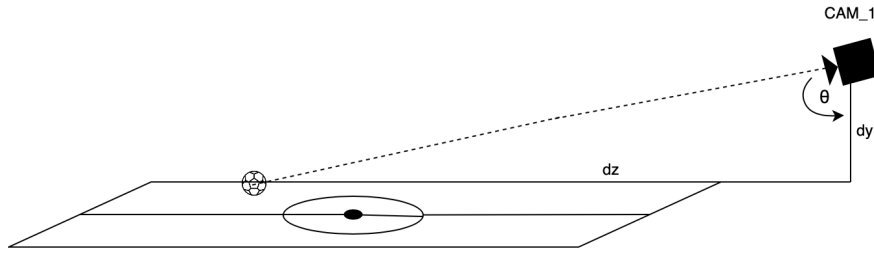


Figure 6.35: Diagram of the tilt angle calculation

$$\theta_{tilt} = -30 + \text{acos}\left(\frac{\text{target}_y - \text{camera}_y}{d}\right) \cdot 0.01 \quad (6.19)$$

The last step in order to successfully track a target is to keep it at a distance where it is easily discernible and to achieve that we need to zoom into the target. To achieve this in a fluid manner an interesting approach was taken. Keeping the same rotation calculations mentioned above, new position calculations are also made. The camera "flies" towards the target up to the set maximum distance (configurable) away from it, the same while moving backwards. The movement is achieved through moving the camera along a ray cast from the target to the camera's initial position. A unitary vector is created along that cast vector and multiplied by the desired distance giving us at the end a vector that originates at the target, points at the original camera position and has a specific length. The final desired position for the camera is obtained by adding the resulting vector to the target's original position (see figure 6.36) and moving the camera to that point.

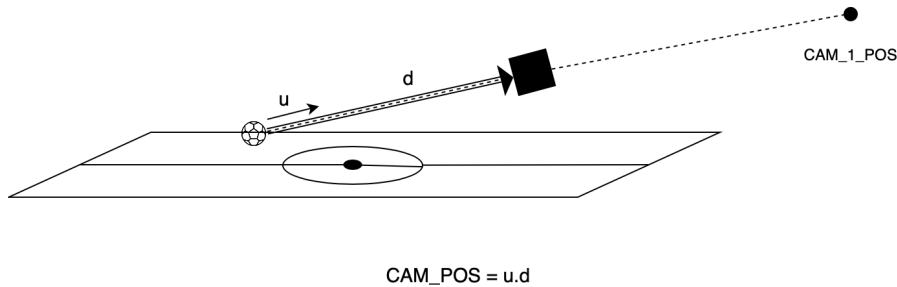


Figure 6.36: Diagram of the camera's position calculation

Once all the necessary transformations have been calculated, they must be applied to the camera. If they would be applied directly, the movement would be abrupt not contributing for a smooth visualization experience nor resembling a real life broadcast feed. To smooth the transition between calculated transformations, the current positions of the camera are "morphed" into the new ones using a *lerp* function (see 6.20). Since the *update()* function runs continuously, this works for a smooth transition between consecutive transforms.

$$\vec{P}_{t+1} = \vec{P}_{final} + ((\vec{P}_t - \vec{P}_{final}) \times \text{scale}) \quad (6.20)$$

At any point in time,  $t + 1$ , the vector for position or rotation is given by the sum of the desired final vector,  $\vec{P}_{final}$  with the difference between itself and the current vector in place,  $\vec{P}_t$ , times a

factor, *scale*, that determines how fast the transition should occur.

Whenever the target is the ball, this *scale* factor is calculated based on its speed to give a more fluid transition given the changes in acceleration that a ball experiences during any type of movement. This way the camera is able to closely mimic a real camera operator that follows with the camera behind the ball at the start of a shot (slow reaction time) and then adjusts towards the end of the movement. This is achieved by applying a factor, *VEL\_SCALE\_FACTOR*, to the real velocity of the ball on screen, *screenVel<sub>ball</sub>* and using its difference from the unitary value (see 6.21).

$$scale = 1 - screenVel_{ball} \times VEL\_SCALE\_FACTOR \quad (6.21)$$

If at the update cycle, the target's instance is of a Player, the behaviour changes slightly. The *scale* factor becomes fixed to provide a more snappy camera movement to follow the player actions without delay, and the maximum distance kept by the camera is decreased to give a close up shot of the player that was marked to be targeted (meaning that he is performing an important action).

#### 6.1.6.2.2 Event Handling

By also implementing the *StatisticsParserListener* interface and having registered itself as a **observer** to the *StatisticsParser* events, the *LiveDirectorCamera* will be aware of relevant actions on the pitch.

For each of the implemented camera configurations for set pieces, when a set piece event, such as a *goal kick*, is received, the camera type is set to its respective type directly influencing which method is run on the *update()* function. These are the *Direct* or **Discrete** events type.

When it comes to the **Calculated** or **Continuous** events type a different approach is taken. Continuous events do not have a single point in time when they occur, instead, as the name indicates, they have a start and an end point, giving us a time frame of its occurrence. This is true for the event driven statistics calculated by the *StatisticsParser*, such as the *dribble detection* and the *shot detection*. For cases like this, the start and stop triggers that indicate each respective start and end points in time and, for each, the *LiveCameraDirector* class implements the updates it sees fit in order to manipulate the camera feed accordingly.

At the reception of a *dribble start* event, the class's target is set to be the player that was identified as the dribbler and the maximum distance of the camera from its target is decreased. At this point, the camera focus on the player and gets closer to provide a better coverage of its actions. The camera resumes its normal broadcast if one of two events are detected, either the dribble has finished (the player has lost the ball) or a shot was detected and the emphasis must be placed on the ball. These events are translated through the implemented interface by the methods *dribbleStopReceived* and *shotStartReceived()*, respectively. Both resume the target has being the ball, the main difference is on the maximum distance that is set, since a *dribble stop* means a normal resume of the game (wider field of view of the pitch) and a *shot start* requires a close up on the ball to follow its trajectory into the back of the net (hopefully).

## 6.2 Conclusions

In this chapter, the development of this project's work was described by explaining thoroughly the implemented architectural decisions and each of its moving parts. The *StatisticsParser* class was presented, where all the heavy-lifting of the data processing is accomplished, enabling the abstraction of the statistics collection process and its access to any party interested either in the collected data or the detection of events. Afterwards, an explanation of how the relevant data was rendered in a structured manner on the UI was made and then how the camera control was developed in order to provide a more realist soccer broadcast feel by being aware of the match's events. It was a solution that in the end involved several fields of study such as: graphical computation, cinematography, programming design patterns, camera control and data structures.

The next chapter illustrates the results obtained by external evaluations of the developed software described in this chapter.

## Chapter 7

# Result Validation Analysis

In order to validate the possible contributions of this work to the RoboCup community and to the RoboViz application overall, an anonymous online questionnaire was created and distributed to individuals of very distinct fields and to general population. The questionnaire had an initial personal characterization question (see 7.1.1) in order to identify the level of familiarity of the participant regarding the concepts around this work and was then divided into two main sections with several questions each, a section dedicated to validation of the *StatisticsOverlay* class and another on the *LiveDirectorCamera* class (see 7.1.2). Within each question there was either an image or a video where a specific feature was presented by ways of demonstrating the RoboViz application before and after the development of this work, and then the participant was asked to rate several statements from 1 to 5, where 1 would indicate that the participant strongly preferred the original version of the visualizer, reveal a neutral position and 5 expresses a strong preference for the newly developed version.

### 7.1 Survey

This section breaks down the survey developed to evaluate the developed modules. The survey can be found here: <https://forms.gle/hZjideWB7FFTP6Zm9>.

#### 7.1.1 Participant Characterization Questions

Participant characterization questions are identified through this work by  $C_n$ , they evaluate the participant's knowledge with a given concept from 1 to 5 (see 7.1).

#### 7.1.2 Classification Questions

The survey questions, identified by  $Q_n$ , were single questions accompanied by a video or image and several statements which the participant evaluated from 1 to 5.

Table 7.1: Participant Characterization Questions

ID	Question
C1	Soccer Concepts
C2	Video Games Concepts
C3	Multi-agent Systems
C4	Robotics
C5	The RoboCup initiative
C6	RoboCup Soccer Simulation League 3D

### 7.1.3 Q1 - Timeline

#### 7.1.3.1 Question

"A timeline overlay was created on the bottom right corner of the visualizer to display the current game completion through an animated bar and goal occurrences."

#### 7.1.3.2 Statements

- Readability of the current game completion
- Readability of the game event's time
- Realistic broadcast

### 7.1.4 Q2 - Heatmap

#### 7.1.4.1 Question

"A heatmap overlay was created on top of the existing positions overlay of the visualizer to display the frequency of the ball position over time, giving a sense for the most played zones."

#### 7.1.4.2 Statements

- Readability of the game flow
- Readability of the game events in time
- Realistic broadcast
- Info on team performance

### 7.1.5 Q3 - Statistics Panel

#### 7.1.5.1 Question

"Similarly to live soccer, a statistics panel was created to be displayed on the top right corner with the stats on a given event that was triggered, like dribbles for example."

**7.1.5.2 Statements**

- Readability of the game flow
- Readability of the game events in time
- Realistic broadcast
- Info on team behaviour

**7.1.6 Q4 - Possession Graph****7.1.6.1 Question**

"Periodically, an overview of the teams' ball possession and possession history is shown through a common percentage number side-by-side and a history graph."

**7.1.6.2 Statements**

- Readability of the game flow
- Readability of the team's performance over time
- Realistic broadcast

**7.1.7 Q5 - New Cameras****7.1.7.1 Question**

"The new cameras have a smoother animation and the realistic feel of an anchored camera at the midfield line, has in real-life soccer."

**7.1.7.2 Statements**

- Readability of the game flow
- Realistic broadcast
- Smooth visualization experience

**7.1.8 Q6 - Set Cameras****7.1.8.1 Question**

"The new set of cameras have specific positions for distinct stopped ball events."

#### **7.1.8.2 Statements**

- Readability of the game flow
- Realistic broadcast
- Smooth visualization experience

#### **7.1.9 Q7 - Timely Possession**

##### **7.1.9.1 Question**

"Periodically, the possession stats are shown with access to history. Also, the camera follows to the side of the pitch like the real spider-cameras hang from the top."

##### **7.1.9.2 Statements**

- Readability of the game flow
- Realistic broadcast
- Smooth visualization experience
- Info on team behaviour

#### **7.1.10 Q8 - Player Follow**

##### **7.1.10.1 Question**

"Events like dribbles and shot preparations trigger emphasis on the player. At the trigger of an event (like fouls) the respective overlay is triggered to give access to a count of events. A shot is more visible on camera."

##### **7.1.10.2 Statements**

- Readability of the game flow
- Realistic broadcast
- Smooth visualization experience
- Info on team behaviour
- Oversight of close-encounters between players
- Better viewing angle for shots



### 7.1.11 Q9 - Dangerous Plays

#### 7.1.11.1 Question

"Dangerous plays around the goal with opponent's possession get extra attention from the camera's director, zooming in on 1-on-1 and out on inwards movements."

#### 7.1.11.2 Statements

- Readability of the game flow
- Realistic broadcast
- Smooth visualization experience
- Better viewing angle for plays
- More exciting experience

## 7.2 Analysis

As of now, the questionnaire as a total of 49 validated answers collected through a span of 15 days. We will first analyze the level of knowledge within each specific topic related to this thesis work, to better understand our participants.

Please evaluate your knowledge on the following topics (1 - no knowledge, 5 - proficient)

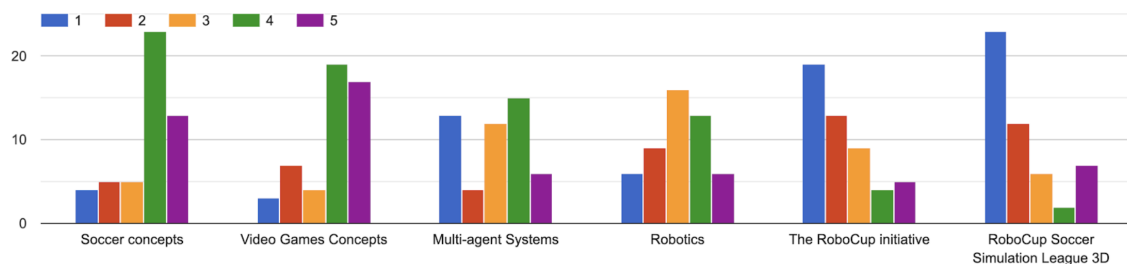


Figure 7.1: Topic classification analysis

Figure 7.1 represents the results of the characterization of the participants. The overall respondent has a good level of knowledge on the fields of robotics and multi-agent systems and the vast majority has a strong degree of familiarity with either soccer or video-games concepts. However, a small part of the participants had a sufficient level of awareness of the RoboCup initiative and more specifically of its Soccer Simulation League 3D. This was somewhat expected due to the sheer difference in the amount of emails gathered between general population and RoboCup participants to send the form.

Since all questions had the same numerical scale ordered from 1 to 5 in the direction of a strong preference for the original version to a strong preference to the new version, the overall success of

the work developed, measured by the validation questionnaire, is given by a larger total count of the levels 4 and 5 of the responses, which directly state a preference for this work's improvements. The pie-chart represented in figure 7.2 shows a clear preference for the newly improved RoboViz visualization experience by these two levels (4 and 5) collectively accounting for 74.6% of the total questionnaire responses.

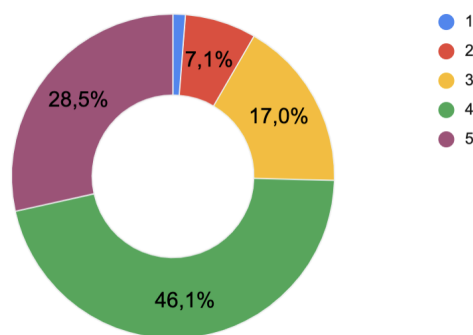


Figure 7.2: Overall classification of the developed solution

The same can be done to individually evaluate the *StatisticsOverlay* and *LiveDirectorCamera* independent of each other. This gives a better sense for the distribution of the results enabling to detect which module contributed more for the positive overall classification of the developed solution.

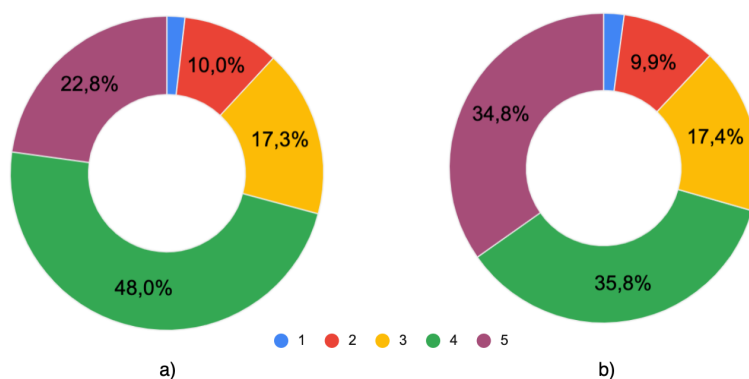


Figure 7.3: Individual classification of the a) *StatisticsOverlay* and the b) *LiveCameraDirector*

Figure 7.3 shows the individual results for the *StatisticsOverlay* class and the *LiveDirectorCamera*, respectively. From the analysis we can see some interesting results, such as a confirmation that both of the developed modules have nearly the same classification percentages for the rates of 1, 2 and 3, meaning that neither of the modules had a substantial impact on the negative ratings received however, the contrary is not true. The camera improvements received a far better evaluation compared directly to the overlay module on the same rating level of 5, the highest. Having almost exactly the same percentage values in the sum of the classifications of rating levels

4 and 5 (around 70% combined), the difference relies only on their distribution among the levels, leaving the camera module to take the win with the most level 5 classifications.

### 7.3 Spearman's Rank-Order Correlation

In order to better understand the relationship between the participant's characterization and its influence on each question's classification, a Spearman's rank-order correlation analysis was made on the data. This allows the measurement of statistical dependence between the classification of the two ordinal variables, participant's characterization question classification and module evaluation's question classification.

Our goal is to analyse whether there is some degree of correlation between a given characteristic of the participant and the given answers per topic, either negative or positive, meaning that our research question is 2-tailed. In this case, our *null* and *alternative* hypotheses,  $H_0$  and  $H_a$ , respectively, can be defined, in terms of the population correlation,  $\rho_s$ , as:

$$H_0 : \rho_s = 0 \quad (7.1)$$

$$H_a : \rho_s \neq 0 \quad (7.2)$$

To compute the test statistic,  $t$  – value, we calculated the Spearman's correlation coefficient,  $r_s$ , using the defined equation for non-distinct integer ranks, with (see 7.3):

$$r_s = \frac{cov(rg_X, rg_Y)}{\sigma_{rg_X} \sigma_{rg_Y}} \quad (7.3)$$

where  $cov(rg_X, rg_Y)$  is the covariance of the rank variables, and  $\sigma_{rg_X}$  and  $\sigma_{rg_Y}$  are their standard deviations. After the computation of the correlation coefficient we calculate  $t$  – value with the resulting  $r_s$  and the number of ranks,  $n$  (see 7.4):

$$t = \sqrt{\frac{n-2}{1-r_s^2}} \quad (7.4)$$

It is then calculated the proof value,  $p$  – value, by constructing a  $t$  distribution with  $n - 2$  degrees of freedom. The  $p$  – value will be the area under the resulting  $T$  distribution that is more extreme than the  $t$  – value observed previously, in the direction of the alternative hypotheses,  $H_a$ .

Finally, the resulting  $p$  – value is considered against the alpha level,  $\alpha$ , considered to be 0.05. If  $p < \alpha$ , we reject the null hypothesis and can state that there is evidence of a relationship between that specific participant's characteristic classification and the question classification, the correlation is significant.

Since each individual question has two or more statements that can be classified from 1 to 5, the analysis was made with the overall classification of the question, which is the sum of all

statement's given grade. Then, the aforementioned process was executed for each characterization question global score.

Table 7.2 show the results of the correlation analysis made on the survey answers. Marked by yellow are the  $p$  – values bellow the  $\alpha$  level and in green are the correlation coefficient considered relevant by the consequent rejection of the null hypothesis by  $p$  – value. The results show that there are some significant positive correlations for four of the five characterization questions, signifying that the higher the familiarity of that given concept, the better the developed modules are classified. They also show no evidence of significant negative correlations which indicates that survey participants more experienced in this project's main concepts does not result in a lower classification of the improvements.

Table 7.2: Spearman's Rank Order Correlation Results

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
C1	$r_s$	0.030	0.205	0.200	0.298	0.204	-0.099	0.283	0.267	0.086
	$p$ – value	0.840	0.157	0.169	0.038	0.159	0.499	0.049	0.064	0.557
C2	$r_s$	0.205	0.075	0.318	0.238	0.469	-0.066	0.345	0.275	0.061
	$p$ – value	0.157	0.609	0.026	0.100	0.001	0.653	0.015	0.056	0.677
C3	$r_s$	0.138	0.361	0.066	0.218	0.108	-0.109	0.218	0.053	-0.061
	$p$ – value	0.346	0.011	0.650	0.133	0.461	0.455	0.132	0.717	0.680
C4	$r_s$	0.140	0.228	-0.121	0.075	-0.076	-0.159	0.114	-0.145	-0.074
	$p$ – value	0.338	0.115	0.408	0.609	0.602	0.275	0.434	0.320	0.614
C5	$r_s$	0.301	0.420	0.020	0.176	0.075	0.077	0.295	0.093	0.201
	$p$ – value	0.035	0.003	0.892	0.227	0.606	0.600	0.039	0.527	0.166
C6	$r_s$	0.364	0.446	0.052	0.186	0.095	0.033	0.275	0.080	0.171
	$p$ – value	0.010	0.001	0.724	0.201	0.517	0.822	0.056	0.584	0.239

## Chapter 8

# Conclusions and Future Work

Besides all the conclusions presented at the end of each chapter, we can say much more on the basis of this work. The RoboCup initiative has taken a huge role in the pursuing of the development of Artificial Intelligence and has gained such a level of following that each organized event has attendances of several thousands of people yearly. It promotes the creation of a plethora of projects where each individually stimulates the work in many other areas, creating a sort of a beneficial vicious cycle. The RoboViz project, where this work was built upon, is a great example of that large array of fields that are interconnected by the simulation of robotic soccer. It encompasses the fields of Multi-agent Systems, Distributed Artificial Intelligence, Graphical Computation and Cinematography that work together towards the goal of simulating a realistic broadcast of a top-level robotic soccer match.

The development of this work was done with the objective to contribute to that main goal also and tried to improve the broadcast level of the current simulation system by enabling it to react to unique events resulting from the behaviour of the agents individually. Many concepts from the fields of Graphical Computation, Cinematography and Event-Driven Systems were applied to work together in enhancing that visualization experience. On top of that, the development was also done with modern software development theory practices in mind, resulting in an extension of code that can easily be implemented onto the original project and further extended without much integration effort.

Overall, it is felt that the main objectives and requisites for this work were generally achieved with success with the final result being an improved visualizer for the RoboCup Soccer Simulation 3D League which provides a more realistic real-time broadcast with relevant on-screen displays that provide information served just-in-time, serving as a base for future work to continuously take the visualization to the next level.

### 8.1 Future Work

The main goal after the conclusion of this work, is to make it available for all of the RoboCup community to use. To make that happen, the source code of this project would need to be integrated

into the official version of RoboViz which is achieved by cleaning up the code and submitting a pull request to the repository after completing the development. After the implementation of the version described in this work and given the accessible modularity of the final developed solution, there is still room for improvement of current features as well as for the development of new ones. As time goes by, new ideas and relevant features will come to mind that would improve even further the visualization experience. As of now, there are a few that can be listed for future development:

1. Implementation of a configuration overlay that would enable to configure certain aspects at run-time.
2. Implementation of a report generation at the end of a match with the collected statistics.
3. Improvement over the current live feed camera orientation regarding which is the attacking team.
4. Further development of the responsive behaviour between screen size regarding the overlays.
5. Implementation of the detection of an attack attempt to be listed as a statistic.
6. Implementation of the detection of successful and failed passes.
7. Further development of the heat map to optionally show an individual player's field occupation.
8. Implementation of an overlay that shows team formation over time.

Taking these future development ideas into account and many more that are yet to come, this extension of the current RoboViz application could become much more powerful in terms of providing the researches with insightful information at the same time as it produces a more realistic viewing experience to the spectators.

# References

- [1] E. Andrew and THUGCHilDz. National geographic society map june 2006 edition "soccer united the world", (accessed June 13, 2020). <https://commons.wikimedia.org/wiki/File:Popularsports.PNG>.
- [2] RoboCup Association. What is robocup?, (accessed June 14, 2020). <https://2020.robocup.org/en/what-is-robocup/>.
- [3] Multiple authors. Episkyros, (accessed June 13, 2020). <https://en.wikipedia.org/wiki/Episkyros>.
- [4] W. Bares and J. Lester. Intelligent multi-shot visualization interfaces for dynamic 3d worlds. In *Proceedings of International Conference on Intelligent User Interfaces*, 1992.
- [5] J. Blinn. *A Trip Down the Graphics Pipeline*. Morgan Kaufmann, 1988.
- [6] A. Bond and L. Gasser. *Reading in Distributed Artificial Intelligence*. Morgan Kaufmann Pub, 1988.
- [7] R. Brooks. Intelligent without reason. In *Twelfth International Joint Conference on Artificial Intelligent (IJCAI-91)*, pages 569–595. Sydney, Australia, 1991.
- [8] P. Burelli. *Game Cinematography: From Camera Control to Player Emotions*, pages 181–195. Springer International Publishing, Cham, 2016.
- [9] J. Bödecker and M. Asada. SimSpark – Concepts and Application in the RoboCup 3D Soccer Simulation League. *Semantic Scholar*, 2008.
- [10] J. Castelo. *Guia Prático de Exercícios de Treino*. Lisboa: Visão e Contextos, 2003.
- [11] J. Collomosse. *Fundamentals of Computer Graphics - CM20219*. CreateSpace Independent Publishing Platform, 2014.
- [12] Pedro Manuel Henriques da Cunha Abreu. *Artificial Intelligence Methodologies Applied in the Analysis and Optimization of Soccer Teams Performance*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2011.
- [13] D. Eck. *Introduction to Computer Graphics*. Hobart and William Smith Colleges, 2018.
- [14] E. Ekaette. Co-ordination in multi-agent systems: An overview. 2002.
- [15] FIFA. Who we are - fifa survey: approximately 250 million footballers worldwide, 2001 (accessed June 9, 2020). <https://www.fifa.com/who-we-are/news/fifa-survey-approximately-250-million-footballers-worldwide-88048>.

- [16] FIFA. Woman's football survey, 2014.
- [17] T. et al Finin. Specification of the kqml agent communication language. *DARPA Knowledge Sharing Initiative External Interfaces Working Group*, 1993.
- [18] L. Guadagno G. Abowd, J. Engelsma and Okon. O. Architectural analysis of object request brokers. *Object Magazine*, pages 44–51, March 1996.
- [19] M. Glavic. Agents and multi-agent systems: A short introduction for power engineers. 2006.
- [20] M. Gleicher and A. Witkin. Through-the-Lens Camera Control. In *SIGGRAPH 92*, July 1992.
- [21] C. Castelfranchi H. Hexmoore and R. Falcone. Agent autonomy. In *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 3–5. Springer Media, 2003.
- [22] Y. Kuniyoshi I. Noda H. Kitano, M. Asada and E. Osawa. Robocup: The robotic world cup initiative. In *Proceedings in Workshop on Entertainment and AI/Alife*, pages 19–24, 1995.
- [23] Y. Kuniyoshi I. Noda H. Kitano, M. Asada and E. Osawa. Robocup: The robotic world cup initiative. In *Proceedings of the International Conference on Autonomous Agents*, pages 340–347. ACM, 1997.
- [24] history computer.com. Sketchpad, (accessed June 11, 2020). <https://history-computer.com/ModernComputer/Software/Sketchpad.html>.
- [25] T. Hoser. *Introduction to Cinematography: Learning Through Practice*. Taylor & Francis, 2018.
- [26] M. Huhns and L. Stephens. Multiagent systems and societies of agents. In Gerhard Weiss, editor, *Multiagent systems: A Modern Approach to Distributed Artificial Intelligence*, pages 79–120. The MIT Press, 1999.
- [27] C. Håkansson. A case study of children of men and clerks ii. *Re-examining the Traditional Principles of Cinematography of Modern Movies*, 2013.
- [28] K. Hiraki I. Noda, H. Matsubara and I. Frank. Soccer server: A tool for research on multi-agent systems. *Applied Artificial Intelligence journal*, pages 233–250, November 2010.
- [29] E. Gomes J. Rocha, I. Boavida-Portugal. *Multi-agent Systems*. IntechOpen, 2017.
- [30] N. Jennings. On agent-based software engineering. *Artificial Intelligence Journal*, 117:277–296, 2000.
- [31] Jojoscope. Kemari: proto-história do futebol, June 2018 (accessed June 13, 2020). <http://jojoscope.net/2018/06/22/kemari-proto-historia-do-futebol/>.
- [32] J. Kim and P. Vadakkepat. Multi-agent systems: A survey from the robot-soccer perspective. *Intelligent Automation and Soft Computing journal*, 6(1):3–17, 2000.
- [33] Sérgio Fernando Grilate Louro. Sistema Multi-Agente para Controlo de Câmaras Inteligentes. Master's thesis, Faculdade de Ciências, Economia e Engenharia Universidade do Porto, 2004.



- [34] E. Foroughi F. Heintz Z. Huang S. Kapetanakis K. Kostiadis J. Kummeneje J. Murray I. Noda O. Obst P. Riley T. Steffens Y. Wang M. Chen, K. Dorer and X. Yin. RoboCup Soccer Server. February 2003.
- [35] T. Nakajima M. Kone, A. Shimazu. The state of the art in agent communication languages. In *Knowledge and Information Systems*, pages 259–284. Springer-Verlag Londo, 2000.
- [36] D. Lukose P. Anthony and C. Kim On. Agent architecture: An overview. In *Transactions on Science and Technology*, pages 18–35, 2014.
- [37] F. Pinheiro and J. Coelho. *A Paixão do Povo: História do Futebol em Portugal*. Simon and Schuster, 1st edition, 2002.
- [38] I. Sethi R. Khosla and E. Damiani. *Intelligent Multimedia Multi-Agent Systems: A Human-Centered Approach*. Spring Media, 2000.
- [39] Ranjith Raghunathan. Blender coordinate system to opengl, (accessed June 10, 2020). <https://www.ranjithraghunathan.com/blender-coordinate-system-to-opengl/>.
- [40] Luís Paulo Reis. *Coordenação em Sistemas Multi-Agente: Aplicações na Gestão Universitária e Futebol Robótico*. PhD thesis, Departamento de Engenharia Electrotécnica e de Computadores, 2003.
- [41] S. Russel and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 1995.
- [42] T. Galyean S. Drucker and D. Zeltzer. Cinema: A system for procedural camera movements. In *SIGGRAPH Symposium on 3D Interaction*. MIT Media Group, 1992.
- [43] L. Lee S. Nwana and N. Jennings. *Co-ordination in multi-agent systems*. Springer Berlin Heidelberg, 1997.
- [44] D. Salomon. *Transformations and Projections in Computer Graphics*. 01 2006.
- [45] M. Singh. Considerations on agent communications. 1997.
- [46] Justin Stoecker and Ubbo Visser. RoboViz: Programmable Visualization for Simulated Soccer. In *T. Röfer et al. (Eds.): RoboCup 2011*. Springer-Verlag Berlin Heidelberg, 2012.
- [47] P. Stone and M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, pages 345–383, 2000.
- [48] K. Sycara. Multiagent systems. *AI Magazine*, pages 79–92, 1998.
- [49] Mowbray. T. Essentials of object-oriented architecture. *Object Magazine*, pages 28–32, September 1995.
- [50] Topendsports.com. Top 10 list of the world’s most popular sports, (accessed June 9, 2020). <https://www.topendsports.com/world/lists/popular-sport/fans.html>.
- [51] Unkown. Raster vs vector, (accessed June 9, 2020). [https://vector-conversions.com/vectorizing/raster\\_vs\\_vector.html](https://vector-conversions.com/vectorizing/raster_vs_vector.html).
- [52] S. Thainimit W. Bares and S. McDermott. A model for constraint-based camera planning. *Spring Symposium Smart Graphics*, 2000.

- [53] J. Wilson. *Inverting the Pyramid*. Orion Books, 2008.
- [54] G. Wong and J. Wang. *Real-Time Rendering: Computer Graphics with Control Engineering*. Automation and Control Engineering. CRC Press, 2017.
- [55] M. Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley & Sons, 2002.
- [56] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review journal*, 10(2):115–152, 1995.
- [57] Yuan Xu and Hedayat Vatankhah. SimSpark: An Open Source Robot Simulator Developed by the RoboCup Community. In *RoboCup 2013*. Springer-Verlag Berlin Heidelberg, 2014.
- [58] A. Souglis Y. Giossos, A. Sotiropoulos and G. Dafopoulou. Reconsidering on the early types of football. *Baltic Journal of Health and Physical Activity*, 3, January 2011.