Augmented Reality Visualization for Autonomous Robots

Danny Zhu

CMU-CS-17-129
December 2017

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the National Science Foundation under grant number IIS-1218932, the Air Force Research Laboratory under grant number FA87501620042, the Office of Naval Research under grant number N00014-09-1-1031, and Microsoft. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: artificial intelligence, augmented reality, autonomous robots, mobile robots, RoboCup Small Size League, visualization

To those who love me.

Abstract

We believe that it is essential to be able to analyze the reasoning of autonomous robots as it relates to their behavior, and to be able to display this reasoning in a quantitatively correct manner. Videos of robots are often naturally used to aid in replaying and demonstrating robot performance; plain videos contain no information about the processing or behavior of the robots, but such videos can be enhanced by being combined with extra information. Overall, the goal of this thesis is to combine real systems of mobile robots with tools for visualizing algorithms, so that the behavior of complex autonomous agents can be displayed in tandem with the real world. Concretely, the thesis will investigate the addition of visualizations onto initially plain, uninformative videos.

We focus on the creation of augmented reality visualizations to explain the reasoning of three autonomous robot systems — a quadrotor, a robot soccer team (with separate discussions of offense and defense aspects), and a robot soccer automatic referee (autoref) — and how to extend those visualizations to general robot domains. After motivating the work by providing a detailed explanation of how to build a visualization of reasoning for an example quadrotor domain, we explain how to generalize the concepts introduced in the process. We contribute a specification of a means of storing a set of graphical information, along with corresponding times and additional organizational information, that we can use to store sets of time-varying spatial drawings and combine them with videos to create visualizations. We also demonstrate a working implementation of the whole procedure, which we have already used with those robots.

An important part of the contribution is the ability to dynamically change what objects are visualized for a given plan. We can show multiple levels of detail of the plan, or filter the visualizations corresponding to different portions of the plan, according to a viewer's choices. Allowing these actions extends the idea of layered disclosure for text, which we have already heavily used, to these visualizations.

After introducing the general contributions, we return to specific robots: the quadrotor again, as well as the soccer offense, soccer defense, and autoref algorithms. For each algorithm, we give a description of the algorithm itself (brief for the offense, detailed for the other two) followed by an exploration of how to instantiate the general visualization principles for that particular algorithm. We demonstrate that we can apply our general methods of visualization across multiple diverse robot systems.

Acknowledgments

The creation of this thesis has been a long process, with many twists and turns; without the steady guidance, inspiration, and prodding of my advisor, Manuela Veloso, I would certainly not have come anywhere near finishing it.

I would also like to thank the rest of my thesis committee: Emma Brunskill, Maxim Likhachev, and Stefan Zickler. Your feedback on my proposal, this document, and my defense was a great help as I refined my ideas and presentations.

And I'd like to thank the wonderful staff of our department, especially Christina Contreras, Sharon Cavlovich, Catherine Copetas, and Deb Cavlovich, for, variously, helping with my travel reimbursements, facilitating our group and individual meetings, securing funding for social events, and reminding me to promptly write my Black Friday letters. Not to mention the thousand other things involved in keeping the students happy, too.

I would like to very much thank those who played a part in creating the complex hardware and software apparatus that CMDragons is, giving me an amazing robot platform to work with over the past several years. In particular, thanks to Joydeep Biswas and Juan Pablo Mendoza for guiding me through the early stages of working with CMDragons and for keeping on top of the many issues to keep it all running and competition-ready while I've been around.

Thanks, too, to all my labmates and the other students I've known during my time here. The club meetings (thank you, Astronomy Club!), board games, and miscellaneous hangouts were always a welcome distraction from whatever I was really supposed to be doing. Thank you, SCS musical, Flutonium, Kiltie Band, Tartan Wind Ensemble, and All University Orchestra, for keeping my musical performance skills sharp through my CMU years.

Finally, I thank my family, nuclear and extended, who have brought me up and supported me through school and work. I wouldn't be anywhere, much less on the receiving end of a PhD, without you.

Contents

1 Introduction **1**

1.1 Approach and contributions 3

1.2 Document outline 6

2 Motivating example: quadrotor navigation **7**

2.1 Setup of autonomous robot experiment 7

2.1.1 Control inputs 7

2.1.2 Output and localization 8

2.1.3 Navigation task 8

2.2 Building a visualization of the robot's internals 8

2.2.1 No visualization 9

2.2.2 Position visualization 9

2.2.3 Environment visualization 9

2.2.4 Execution error visualization 11

2.2.5 Route planning visualization 11

2.2.6 Path visualization 15

2.2.7 Simplified path visualization 17

2.2.8 Additional motion information visualization 17

2.3 Drawing process 20

2.3.1 Masking 20

2.3.2 Coordinate transformation 20

2.3.3 Drawing 21

2.4 Discussion and summary 21

3 Domain-independent visualization **25**

3.1 Robot-independent visualization language 25

3.1.1 Language description 25

3.1.2 Instantaneous and persistent frames 26

3.1.3 Drawing primitives 28

3.1.4 Visualization filtering/levels of detail 28

3.2 Concrete software implementation 29

3.2.1 Viewer software and interface 29

3.2.2 Log storage format 29

3.3 Details of drawing process 32

3.3.1 Masking . . . 32
3.3.2 Coordinate transformation . . . 33
3.3.3 Separation of components . . . 36
3.4 Quadrotor revisited . . . 36
3.5 Discussion and summary . . . 39

4 RoboCup SSL **41**
4.1 RoboCup and the SSL . . . 41
4.2 Physical components and communications . . . 42
4.3 Data acquisition . . . 44
4.3.1 SSL-Vision . . . 44
4.3.2 State filtering . . . 45
4.4 Simulation . . . 46
4.5 Human refereeing . . . 47
4.6 Game structure . . . 47
4.7 Rules . . . 49
4.7.1 Time-based rules . . . 49
4.7.2 Robot-based rules . . . 49
4.7.3 Ball-based rules . . . 49
4.8 SSL visualization: CMDragons offense . . . 50
4.8.1 No visualization . . . 51
4.8.2 Position visualization . . . 51
4.8.3 Zone visualization . . . 51
4.8.4 Action selection visualization . . . 53
4.8.5 Execution visualization . . . 53
4.9 Discussion and summary . . . 56

5 RoboCup SSL: threat-based defense **59**
5.1 Overview . . . 59
5.2 High-level states . . . 60
5.3 Spatial algorithms . . . 61
5.3.1 Stadium intersection . . . 61
5.3.2 Stopping location . . . 61
5.3.3 Velocity estimation . . . 62
5.4 Tasks . . . 62
5.5 Primary defenders and first-level threat . . . 64
5.5.1 Threat location . . . 64
5.5.2 Snapping . . . 65
5.5.3 Defense area margin . . . 65
5.5.4 Task positions for primary defenders . . . 67
5.6 Second-level threats . . . 68
5.6.1 Task positions for blocking shots . . . 69
5.6.2 Task positions for blocking passes . . . 70
5.7 Task assignment . . . 71

5.8 Three or more primary defenders . . . 71
5.9 Defender behavior . . . 71
5.9.1 Closing the gap . . . 72
5.9.2 Guarding the ball . . . 72
5.10 Goalie behavior . . . 73
5.10.1 Corner kicks . . . 73
5.10.2 Diving . . . 74
5.10.3 Kick taking . . . 74
5.11 Defense visualization . . . 75
5.11.1 Open angles . . . 75
5.11.2 Projected ball trajectory . . . 75
5.12 Discussion and summary . . . 75

6 RoboCup SSL: Autoref **79**
6.1 The referee interface: inputs and outputs . . . 79
6.2 Hybrid automaton . . . 81
6.2.1 Hybrid automaton formalism . . . 81
6.2.2 SSL automaton . . . 82
6.2.3 Automaton evolution and the rules . . . 85
6.3 Automaton events . . . 87
6.3.1 Event loop algorithm . . . 87
6.3.2 Event detector implementations . . . 90
6.3.3 Challenges . . . 96
6.4 Results . . . 98
6.4.1 Running simulated games . . . 98
6.4.2 Comparison to human refereeing . . . 101
6.4.3 Anecdotal evidence of reactions from RoboCup 2017 . . . 103
6.5 Visualizations . . . 104
6.5.1 Autoref implementation internals . . . 105
6.5.2 Ball touched and ball out . . . 105
6.5.3 Ball kicked too fast . . . 107
6.5.4 Ball stuck . . . 107
6.5.5 Distance from ball during stop . . . 107
6.6 Discussion and summary . . . 107

7 Related work **111**
7.1 RoboCup SSL game analysis and refereeing . . . 111
7.2 Visualization . . . 113
7.3 Mixed reality . . . 114
7.3.1 Augmented reality . . . 115
7.3.2 Augmented virtuality . . . 116

8 Conclusion and future work **117**
8.1 Contributions . . . 117
8.2 Future work . . . 118
8.2.1 Visualization . . . 118
8.2.2 Autoref . . . 119

A Notation **121**
A.1 Typographic conventions . . . 121
A.2 Common variables . . . 121
A.3 Common functions . . . 121
A.4 Common constants . . . 122
A.5 Data structures . . . 122

Bibliography **123**

List of Figures

1.1 A typical frame from a video of an SSL game, showing an environment containing many fast-moving robots and a particularly hard-to-follow ball.. 2
1.2 The same frame shown in Figure 1.1, with additional informative annotations drawn on top of it. 2
1.3 The two-dimensional view generated by our existing viewer software for the same moment depicted in Figure 1.1. 4

2.1 The quadrotor we used for the navigation demonstration domain, alongside its indoor hull. 8
2.2 A plain image taken directly from a video of the quadrotor performing its navigation task. 10
2.3 A video frame with the quadrotor's current and target locations annotated. . . . 10
2.4 A video frame with the obstacles also annotated, showing the virtual environment in which the quadrotor operates. 12
2.5 A sequence of video frames shown in order, showing how the obstacles can explain the complicated path of the quadrotor as it attempts to reach the blue circle. 12
2.6 A video frame from a different time showing the robot's state after it has illegally crossed an obstacle and is attempting to return to a previous legal location. . . . 16
2.7 A video frame showing the full tree explored by the RRT, visualized as line segments joining each state within the tree to its parent. 16
2.8 A video frame with the full tree removed, leaving only the links between states on the actual final path to the goal. 18
2.9 A video frame showing the path after simplification and the point to navigate toward. 18
2.10 A video frame showing the acceleration command sent to the robot. 22
2.11 Left: A 2-D visualization generated from one run of the RRT, using a domain based on physical obstacles. Right: The result of drawing that visualization onto the corresponding frame from the video of the real robot. 22
2.12 The stages in computing the mask for a frame of a video to properly overlay our visualizations. 22

3.1 A grammar describing the contents of the visualization logs. 26
3.2 A conceptual example of applying the visualization language on top of generic images. 27

3.3 The viewer we developed for our cross-robot visualizations, showing the main graphical view, as well as controls for the video and selection of entities. 31
3.4 The cross-robot visualization viewer showing the quadrotor, with not all entities selected. 31
3.5 A demonstration of the stages in the process of masking. 35
3.6 A demonstration of the process of mapping the coordinates from world coordinates in a log to image coordinates in a frame. 37
3.7 The text representation of the frame of the log corresponding to the drawings in Figure 2.11. 40

4.1 The robots of CMDragons, showing (left) a full robot with a ball on its dribbler, (middle) a robot without its top cover, and (right) without its main electronics board. 43
4.2 A representative moment from the RoboCup 2015 tournament, showing an indirect free kick taken by CMDragons. 43
4.3 A schematic diagram of the components of an SSL game. The two teams, their robots, and the neutral cameras together make up one largely autonomous system. 44
4.4 A visual demonstration of the stages of SSL-Vision processing. 46
4.5 The interface displayed by the refbox when it starts up. 48
4.6 A plain video frame showing a time just before a robot makes a pass. 52
4.7 A video frame showing which team each robot is on, along with pseudocode that generates the drawings in the image. 52
4.8 A video frame showing the zones in use by the blue team for the pass in progress, demonstrating how the zones cover the area around the opponent goal. 54
4.9 A video frame showing the zones and default locations in use by the blue team for the planned pass, as well as the specific reception location for the pass. . . . 55
4.10 A video frame showing a time at the end of the pass, when the receiver has control of the ball and is about to kick it toward the opponent goal. 55
4.11 Visualizations showing the successful planning and execution of passes by our SSL team. 57

5.1 A demonstration of the algorithm for intersecting a ray with the boundary of a stadium. 62
5.2 The dependence of the cost of receiving on ball position; a lighter color indicates a lower cost. 66
5.3 A plot of the primary defender margin distance from the defense area as a function of the distance from the first-level threat to the defense area. 67
5.4 The means of checking whether one defender at the appropriate margin can block the open angle by itself. 68
5.5 The means of positioning a single primary defender within the open angle when it cannot cover the whole angle. 69
5.6 Examples of our defense's response to opponent positions while all defenders are primary defenders. 72
5.7 An example of the coordinated guarding behavior of our defenders. 73

5.8 An example of the visualizations showing the open angles from the ball to our goal. 76
5.9 An example of the visualizations generated by the defense-specific ball velocity estimator when the defense evaluator is using it to override the Kalman filter. . 76

6.1 A simplified representation of the automaton representing a game of the SSL. . 86
6.2 A demonstration of the evolution of the autoref over time in response to the motions of the robots and ball. 88
6.3 A visual description of the LINE collision detection algorithm. 95
6.4 A visual description of the BACKTRACK algorithm. 95
6.5 The specialized referee robot. The chassis is 3-D printed and the components were chosen with low cost in mind; the whole robot cost less than $200. 99
6.6 The results of comparing two versions of our team, one handicapped by having either fewer robots or reduced speed and acceleration. 100
6.7 An example of the visualization resulting from the ball touching a robot and exiting the field. 106
6.8 The moment when the ball exited the field after the time shown in Figure 6.7, showing that the ball indeed continued to the exit point indicated by the line. . 106
6.9 An example of the drawings that could result from the ball being kicked too fast, showing a sequence of observed ball positions in consecutive frames that are too far apart for a legal kick. 108
6.10 An example of the visualization shown when the ball is in the process of becoming stuck during play. The ball is stuck between the two robots in the white circle. 108
6.11 An example of the visualization shown when a robot is too close to the ball while the game is stopped; the red circle shows the minimum allowed distance from the ball when the game is stopped. 110

List of Tables

2.1 The mappings from quantities computed by the quadrotor to drawings as shown in the figures of this chapter. 23

6.1 The referee commands used in the SSL. 80
6.2 The values used as input by the autoref. 80
6.3 The high-level automaton locations of an SSL game. 83
6.4 The differential equations for the continuous variables of an SSL game. 83
6.5 The coarse stages of an SSL game. 83
6.6 The auxiliary variables of the automaton representing an SSL game. 84
6.7 The results of comparing two versions of our team, differing by the presence or absence of a new feature. 101
6.8 Performance of the autoref for ball exit events as compared to human referees at RoboCup 2014. 102
6.9 The fraction of times that the autoref or human referee correctly identified the last team to touch the ball before it exited the field, among the times when it correctly called an out. 103
6.10 The counts of detected goals from the games. The only errors were false positives due to chip kicks. 103

List of Algorithms

1 The basic RRT algorithm. . . . 14
2 The algorithm that computes the commands to send to the quadrotor. . . . 19
3 The algorithm for selecting drawing primitives from a log to draw onto one frame of a video. . . . 30
4 The algorithm for computing a background mask based on an image frame. . . 34
5 The algorithms for drawing a set of shapes onto an image, given the necessary masking and transformation information. . . . 38
6 The algorithm that the defense evaluator uses to compute an updated estimate of the ball's position and velocity when the tracker returns unreliable results. . 63
7 The algorithm for estimating the opponent receiver. . . . 66
8 The algorithm to compute the position to place a secondary defender to block a shot threat. . . . 70
9 The overall event-handling loop algorithm. . . . 90
10 The algorithm for determining whether a the ball has left the field and how to respond if so. . . . 92
11 The algorithm that decides whether the ball has become stuck. . . . 93
12 The algorithm to check if the robots are ready to take a kick to restart play. . . . 97

Notice

By its nature, a thesis about visualization on videos would require the inclusion of videos for full effect; of course, we cannot include the videos in this static document, but we make liberal use of images in the document. Relatedly, if you print this document, please print it in color to ensure that the diagrams can be clearly seen.
Additional visualization materials may be found at the following URL: `https://dzhu.us/thesis`.

Chapter 1

Introduction

Imagine watching a game of robot soccer in the RoboCup Small Size League (SSL), as depicted in Figure 1.1. The game that the robots play is fast, and the ball is small, so it is hard to tell what is happening if one does not already know the game very well. It can be hard even to tell which robots are on each team, as shown by the blue and yellow dots on top. Even if we do know the robots well, there is still a lot that we cannot deduce by just looking at the robots moving. That is true while watching robots in person, but it gets worse when watching a video of the robots that has been captured by some camera somewhere. Then the ball can really get lost in the video and the teams may be even harder to follow. Videos are still very informative, because they are the only way to capture a recording of what actually happened in the world. But would it not be even better if the videos had some extra information about the underlying algorithms that determine the robots' behaviors to help us follow what happened?

That is where this thesis comes in. We have investigated the ability to draw extra information on the videos, as shown in Figure 1.2, to capture and reveal what the algorithms of autonomous robots are doing. Much of the internal state of the robots involves physical locations on the ground, e.g., where the ball is, where to pass to, and where the opponents are. By making the video look as if all of that information had actually been displayed on the ground around the robots, we can make it easier to tell what happened and why the robots did what they did. In this thesis, we break down the interesting information from a robot system into three categories, consisting of the information related to (a) the current state of the world, (b) the future plans of the robots, and (c) the reasons that the robots chose to make a particular decision in the past. Because the robots are controlled by a computer program, it is possible to record exactly what their algorithms are doing at every moment. The methods of this thesis then combine that information with a video by drawing informative objects on it, such as circles to mark locations on the ground or lines to describe a region or path; we keep the original view of the robots from the video, and then add drawings.

Robot soccer teams nowadays are controlled by complicated programs, and must consider a lot of information in order to make sure to be playing competitively. For instance, one common architecture that many teams use, including the team of which the author is a part, is Skills, Tactics, and Plays (STP) [Browning et al., 2005]. Skillsg decide the exact low-level commands for one robot to do in order to do one small, specific thing; tactics put together multiple skills and connecting logic to make a robot fill in a higher-level role for the team; finally, plays describe

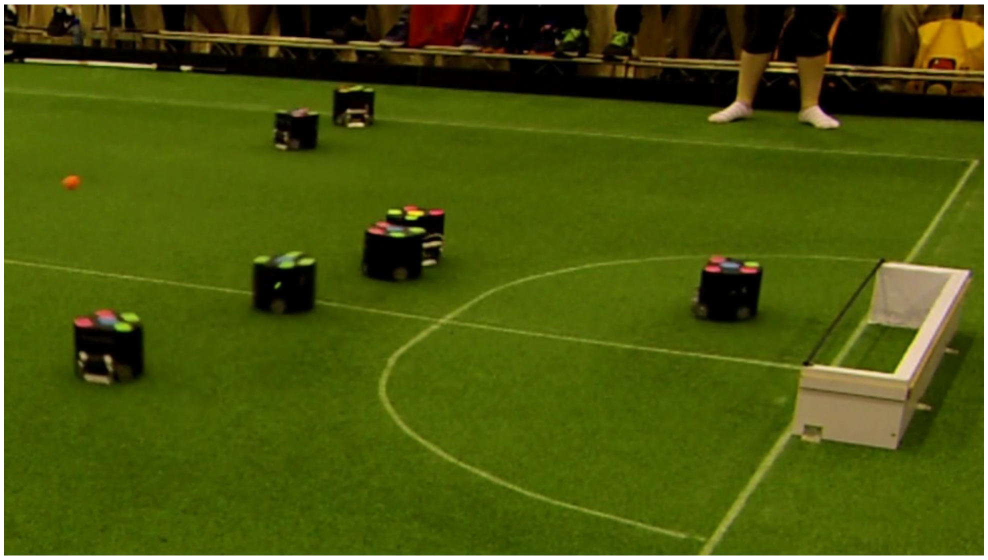


Figure 1.1: A typical frame from a video of an SSL game, showing an environment containing many fast-moving robots and a particularly hard-to-follow ball..

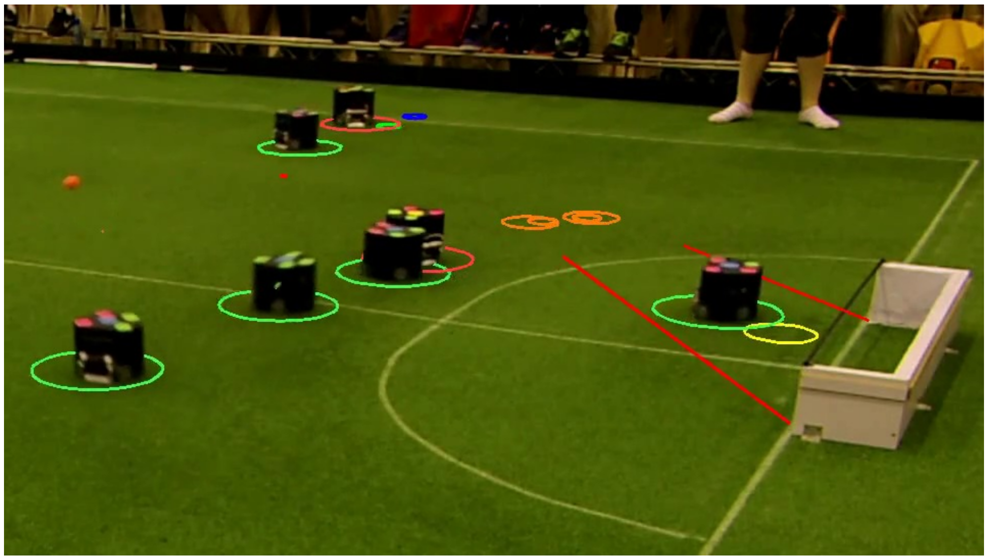


Figure 1.2: The same frame shown in Figure 1.1, with additional informative annotations drawn on top of it.

how to assign tactics to each robot on the team. Every skill, tactic, and play that is active at a given moment has to make decisions constantly in order to figure out the right action to take. Altogether, there is a lot of information that the system generates that might need to be conveyed later on; however, not all of it will necessarily need to be displayed all the time. In this thesis, we produce frameworks that enable the creation of flexible visualizations that make it possible to filter the set of displayed visualizations in a sensible, principled way, and have more or less information visible according to what one might need at any given moment. In order to produce these flexible visualizations, we need to store information during robot execution and be able to interpret it later. In general, this thesis is about taking the execution of a complex robotic system and encode the decision and history information in log files that are later combined with videos.

To be clear, the intended focus of this thesis is not on performing user studies to evaluate visualizations or determine the choices for visualization that lead to the best understanding in any particular situation; rather, we intend to provide the ability to produce the visualizations and make such decisions easily in general situations. We demonstrate feasibility and applicability of

Now, we do need to address the reason that something new like what we have described can and should be used for robotic applications. After all, although there are robots involved, there is ultimately a computer executing a program to control them. We, and other robot developers, do in fact use `print` statements and other text-based displays to an extent, but one immediate problem is that there can be a lot of information to display, so you would need a lot of print statements. Besides, the robots really do change what we need; we can't simply ignore the fact that these programs have physical components in the real world. Any method that draws on a screen in isolation, even if it includes graphical elements, as in Figure 1.3, is separated from the robots themselves, and that gap gets in the way of its ability to convey information. Each part of the program execution is tied to a particular state of the robot in the world, and enabling the display of the direct physical connection with that state.

In summary, this thesis addresses the following question:

> Given a video of autonomous robots, how can information about the decisions and actions of the robots be extracted from a representation of their reasoning and flexibly projected onto the video?

1.1 Approach and contributions

In this thesis, we explore several aspects of the problem of generating informative visualizations from records of robot execution. We create videos that not only show directly what the robots are doing, but also depict inferences about the actions and the reasoning behind them. As a necessary part of the process, we devise ways of representing and presenting the stored data in a way that allows for flexibility in display and usage.

Augmented reality visualizations The visualizations we have automatically created are properly registered with the locations of objects in the world and are subject to occlusion by objects in the scene. In order to create such visualizations, we must therefore have three main pieces of information: the mapping from spatial coordinates in the world to image coordinates in the video, enough understanding of the structure of the scene to handle the occlusion, and

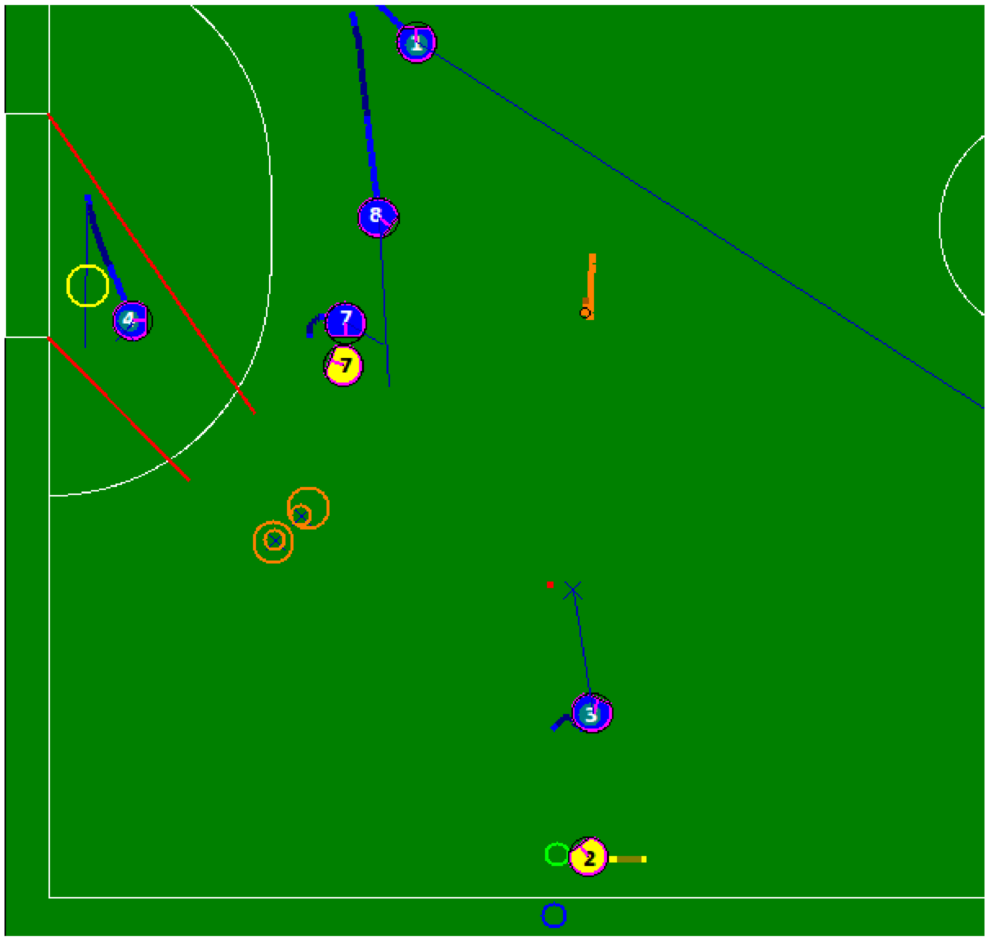


Figure 1.3: The two-dimensional view generated by our existing viewer software for the same moment depicted in Figure 1.1.

additionally the mapping between the times recorded in the execution logs and time in the video, so that all events are synchronized.

For the cases we have examined, we have been able to generate this information manually or with straightforward means. For the case of planar drawings (where the plane is usually the floor), the coordinate mapping is given by a homography [Forsyth and Ponce, 2003]. In general, a homography is determined by four or more point or line correspondences between two coordinate systems; we have considered cases of a static camera, where the homography remains constant throughout the video; for such videos, we supply the correspondences manually.

The occlusion information is also straightforward to determine in some cases, but not so in general. For the situations we have implemented so far, we have only needed to detect the pixels of a distinctively-colored floor; we have used color thresholding followed by morphological operations to do so. For other cases, e.g., if we were using three-dimensional visualizations, we would need more detailed information.

Finally, we need the mapping from the times recorded in the logs of execution (usually given in some absolute scale, such as seconds since the Unix epoch) and time in the video (which starts at 0 at the first frame of the video and increases a constant amount for each frame, based on the frame rate), which is always an additive offset. We determine the offset manually for each visualization.

We categorize the displayable information for any given robotic system into three groups, based on its relation to the current time. One category contains information directly related to the *present*: tracking or sensor information that makes up the current estimate of the state of the world. Then there is the information related to the *future*: the plans of the robots that have yet to be executed. Finally, there is the information from the *past*, showing events from previous windows of time that have some effect on the decisions in the present. The different robot systems that we use to demonstrate visualizations naturally contain information from the different categories.

Multiple levels of detail Given the large amount of information that we might want to display for a run of a robot system, we must be careful not to overload someone looking at the system by displaying too much information.

The concept of *layered disclosure* [Riley et al., 2001] addresses this kind of problem in text-based logging systems. It describes tagging each individual piece of information within a log with a “layer”, which is a numeric value that indicates the abstraction level to which that information belongs (e.g., overall, abstract goals and immediate sensory perceptions at immediate ends of the spectrum, with intermediate, short-term goals in the middle), with selective display and the elements placed into a meaningful hierarchy. We apply a generalization of layered disclosure to the graphical elements in our visualizations, allowing for the selection of not only linearly-ordered levels of importance, but also grouping by other criteria, such as location within the program or relevance to a high-level goal of the robotic system.

Implementation and evaluation on real robot systems In order to demonstrate the feasibility of the algorithms and systems we develop, we have created and tested multiple applications of the visualizations on different robot systems. We have worked with a quadrotor navigation domain and three different algorithms relating to the RoboCup SSL: an offense and a defense soccer algorithm, as well as an automatic referee system. The SSL robots in particular already store and use a rich set of graphical visualization primitives, but their display is simplistic and lacking in the extra capabilities detailed above.

We created a log specification and implementation that apply to all of the robot domains, and could be used with any other, demonstrating the flexibility and applicability of our visualization concepts and allowing us to reuse code across robots. Doing so also enables others to create similar videos for any other robot system; the concrete algorithms we have developed can be used in existing projects to allow the development and use of visualizations with a minimum of additional effort.

1.2 Document outline

The thesis document is arranged as follows.

- **Chapter 2** (**Motivating example: quadrotor navigation**): We present a robot domain and algorithm based on flying a quadrotor that we use to motivate the creation of the visualizations we discuss in the rest of this thesis.
- **Chapter 3** (**Domain-independent visualization**): We describe a generalization of the concepts we introduce in Chapter 2, extending the ideas to a general process for visualizing spatial information on videos.
- **Chapter 4** (**RoboCup SSL**): We introduce the RoboCup SSL, which is the setting for the robot domains we introduce in the following chapters, and describe an offense-related planning algorithm in the SSL.
- **Chapter 5** (**RoboCup SSL: threat-based defense**): We describe an algorithm for the defensive component of an SSL team, primarily developed by the author over the course of the thesis, that has been extensively evaluated in competition scenarios.
- **Chapter 6** (**RoboCup SSL: Autoref**): We formalize and describe an algorithm for an automated referee for the SSL, as well as experimental results from using it.
- **Chapter 7** (**Related work**): We discuss previous work that relates to this dissertation.
- **Chapter 8** (**Conclusion and future work**): We conclude the thesis by summarizing its contributions and discussing possible directions for future work.
- **Chapter A** (**Notation**): We define some of the notational conventions used in the pseudocode listings of the document.

Chapter 2

Motivating example: quadrotor navigation

Our very first motivation to create the autonomy visualizations we have introduced came from the desire to illustrate the behavior of a quadrotor executing a navigation algorithm; accordingly, we begin by using this chapter to describe that domain in depth. We begin by introducing the mechanics of the quadrotor and the task we set for it in Section 2.1. In Section 2.2, we go through the process of building up and motivating the creation of a visualization to explain the operation of the quadrotor, and in Section 2.3, we discuss the process of actually creating the drawings on videos given drawing information.

2.1 Setup of autonomous robot experiment

For this domain, we used a commercially-available ARDrone 2.0 quadrotor, shown in Figure 2.1 along with a protective hull meant to shield the blades of the quadrotor when it is used indoors. In this section, we describe the information we receive from the execution of the robot and the information we ultimately compute to control it.

2.1.1 Control inputs

We control the quadrotor by connecting to it over Wi-Fi from an offboard computer and sending it commands containing desired values for (a) pitch and roll, (b) rate of change of yaw rate, and (c) vertical speed. The pitch and roll values correspond directly to the horizontal acceleration of the quadrotor, since the propellers can only point perpendicular to the body of the quadrotor, so we treat the commands as describing acceleration. We focused on using those values to control the location of the quadrotor above the ground, and used the vertical speed and yaw controls only to keep the quadrotor's altitude and yaw approximately constant.

We consider only sustained flight of the quadrotor, excluding control during takeoff or landing. The built-in software of the quadrotor includes automatic control of takeoff and landing. We send only a signal that the quadrotor should take off from the ground, and it autonomously begins flying and stabilizes itself in the air. Once it is ready to fly, according to the onboard

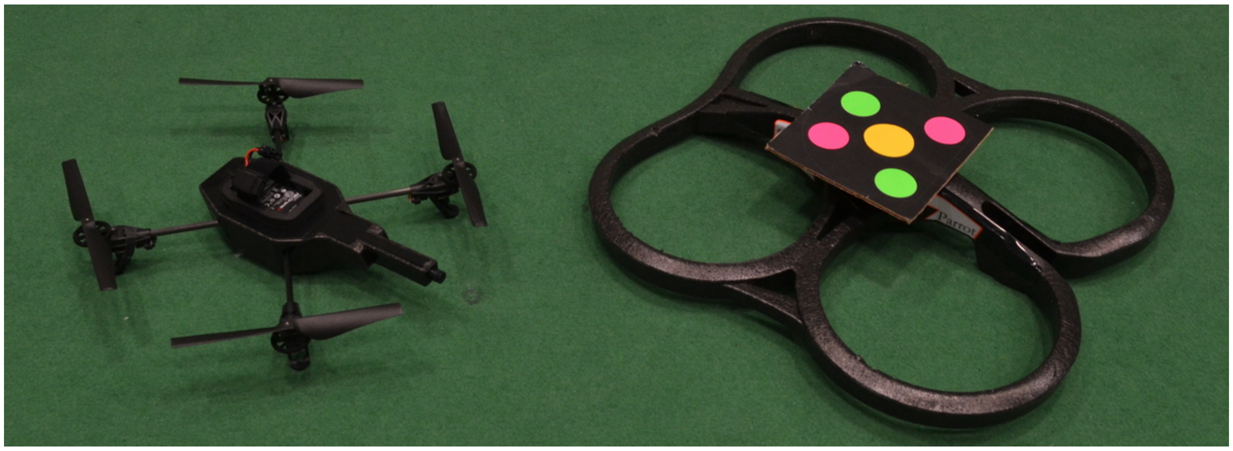


Figure 2.1: The quadrotor we used for the navigation demonstration domain, alongside its indoor hull.

software, it begins responding to acceleration commands. Similarly, when we are done flying, we send a signal to land, and the quadrotor autonomously lands itself safely on the ground.

2.1.2 Output and localization

The colored pattern shown attached the hull in Figure 2.1 is for localization using our preexisting SSL-Vision setup (described in detail later in Section 4.3). Over the same Wi-Fi connection used for control, the quadrotor transmits altitude readings from an ultrasonic sensor on its underside at 15 Hz. The position that SSL-Vision reports is incorrect, since it expects the pattern to be closer to the ground, but we can correct for that effect using the altitude value. We send a control command to the quadrotor after receiving each altitude reading.

2.1.3 Navigation task

As an initial testbed for demonstrating our visualizations, we designed a navigation task taking place in a two-dimensional space with static obstacles. We selected a region within our SSL field for the quadrotor to operate in and manually created *domains*. Each domain consists of a list of points, expressed in the coordinates used for the SSL field, as well as a list obstacles, where each obstacle is a line segment specified by its two endpoints. The task was to make the quadrotor navigate to each target point in turn, starting with the first in the list and cycling back to the top of the list after reaching the end, while not passing through any of the line segment obstacles. If, due to execution error, it passes through an obstacle at a point p, we require it to move to a point q such that the line segment from p to q does not pass through any obstacles before continuing any further.

2.2 Building a visualization of the robot's internals

In this section, we will incrementally describe how to go from a plain video of execution to an annotated one that describes the internals of the robot, at the same time that we specify the calculations that go on during the execution of the quadrotor task. We will introduce the multiple possible behavior autonomy aspects that can be visualized and the ones we have selected, including the justification for our choices. As a running example, we work with one particular run of the navigation task.

2.2.1 No visualization

First, we start with an unannotated image, shown in Figure 2.2. We can see the robot moving (at least when viewing the actual video, rather than a single frame), but we cannot see where the robot is going; the internals of its computations are invisible from the video. The same problem arises when watching any robot; the final observable behaviors of the robot cannot possibly convey everything about the internals.

2.2.2 Position visualization

A natural first question that arises regarding the actions of a mobile robot is:

Where is the mobile robot going?

While the robot's destination is not visible at any time, it is clearly an important part of the internals of the robot. Therefore, we should display the destination of the quadrotor on the video. Similarly, we display the robot's notion of its own current location; although it is (ideally) the same as its real location, drawing the location allows us to verify that the location is in fact correct and, even if it is already correct, puts it in the same spatial context as the other drawings. Figure 2.3 contrasts with Figure 2.2 by showing the same video frame, but where our visualization algorithm has automatically added the robot's current location as a green cross and the robot's destination as a large blue dot.

In general, the destination of the mobile robot may be changing over the time as a function of changes in the environment, as well as possibly the robot's planning to achieve its objectives. The visualization of the relation between the robot's location and its destination provides the basic core understanding of the motion of the robot.

2.2.3 Environment visualization

When the robot starts moving with a visible destination, we can see that the quadrotor is attempting to eventually reach its target point; a question then arises with respect to the specific observed trajectory:

Why doesn't the quadrotor move in a straight line toward its target?

The environment does not have visible obstacles that justify the quadrotor's twisty trajectory. However, in this example, the robot is planning a trajectory to avoid virtual obstacles: obstacles with defined coordinates that do not necessarily correspond to anything in the real world. We defined several different sets of obstacles within the space, where each obstacle is a line segment. Some of the domains were based on real objects, as shown in Figure 2.11, while some were purely virtual, as shown in the running example of this chapter. From the robot's point of view, the environment has obstacles that are not visible within the plain video. We thus add the virtual

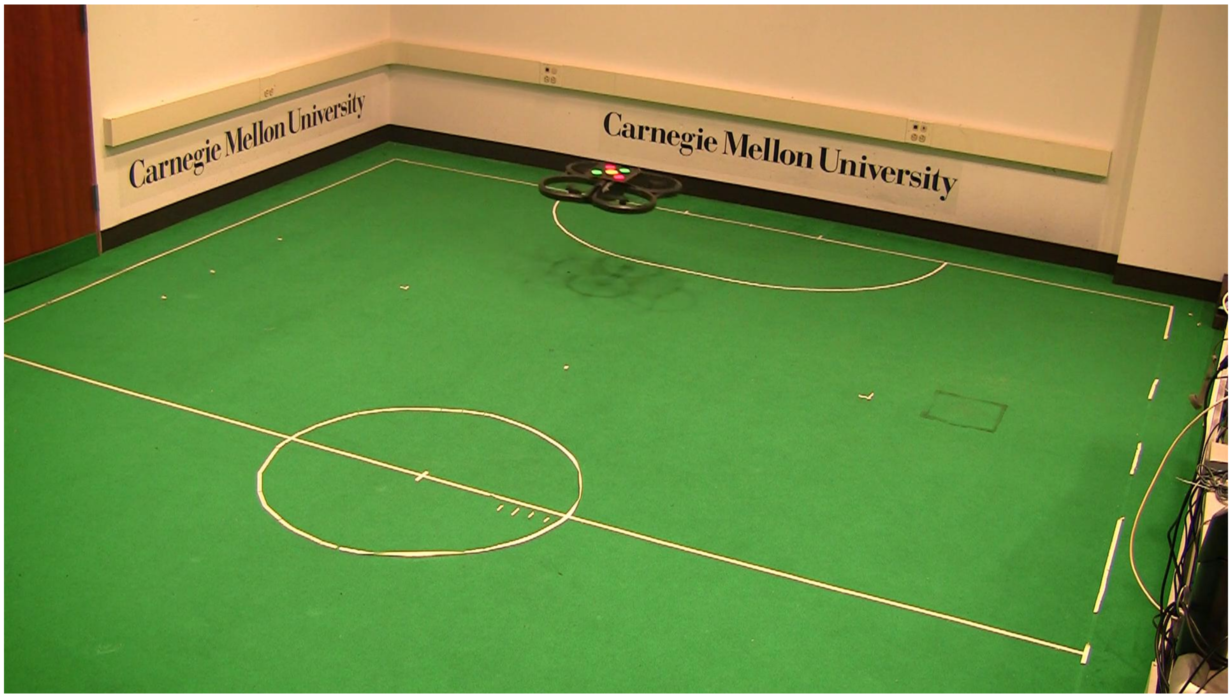


Figure 2.2: A plain image taken directly from a video of the quadrotor performing its navigation task.

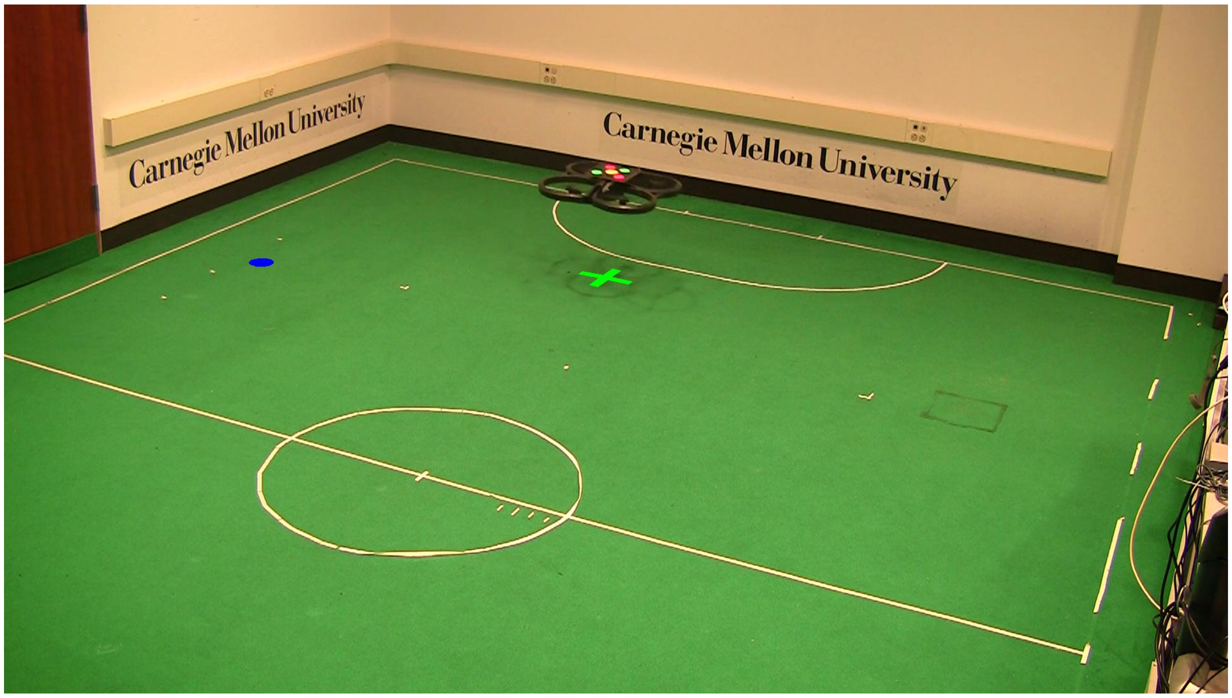


Figure 2.3: A video frame with the quadrotor's current and target locations annotated.

obstacles on the video to visualize the environment as seen by the robot. Figure 2.4 shows the visualization of the target plus the obstacles and Figure 2.5 shows a short sequence of locations of the robot that are now clear to understand in terms of the shown obstacles.

These virtual invisible obstacles capture the general issue of aspects of the environment that an autonomous robot may take into account in its reasoning that are not visible, directly or otherwise, as part of the real world. An example of information that would be partially visible from the real world could be a robot with a configurable minimum distance to keep from physical detected obstacles. Then the visible objects would provide incomplete information about the quantities influencing the robot's reasoning, while drawing inflated outlines of the objects that account for the margin would more completely explain the reasoning.

2.2.4 Execution error visualization

Due to the nature of the problem as we defined it, it is possible for the quadrotor to perform an illegal move by allowing its position to cross directly over an obstacle. In that case, we require that the quadrotor recall the last location it had before first crossing any obstacles and attempt to go straight back to that point, and only resume normal navigation once a line segment from its location to that point crosses no obstacles. With virtual obstacles, this requirement leads to yet another behavior that causes the quadrotor to behave erratically for no visible reason. Figure 2.6 shows how we choose to visualize the robot in this state: a red cross marks the last valid position, toward which the robot is trying to return.

That leads to a general question of

What undesirable actions has the mobile robot taken?

Much of our discussion will focus on the rapidly-changing, quantitative aspects of a robot's execution, but we can also consider larger-scale, qualitative changes in what the robots are doing. In this case, there is a distinct difference between when the quadrotor is actively moving toward its target location and when it is instead only attempting to return to a previous location. In general, robots may have their high-level goals change, and these changes can lead to shifts in which quantities are computed, not only the values of the ones that are.

2.2.5 Route planning visualization

The addition of obstacles, so that we can see the constraints on the quadrotor's motion and get a sense of why it takes complicated paths, might be enough to satisfy some, but there is still a gap between what we can see and what the robot is computing; namely, the details of the actual path planning being done by the robot and the computation of the final commands it executes. The next question that we might want to begin considering is:

How does the robot plan what to physically do?

In order to compute paths through the space, we used the rapidly-exploring random tree (RRT) algorithm [LaValle, 1998], which is a search algorithm that finds a continuous path from a start state to a goal state[1] within some state space that possibly contains impassable obstacles. Each

[1]The RRT algorithm can easily handle a set of goal states, but we use only a single goal state here.

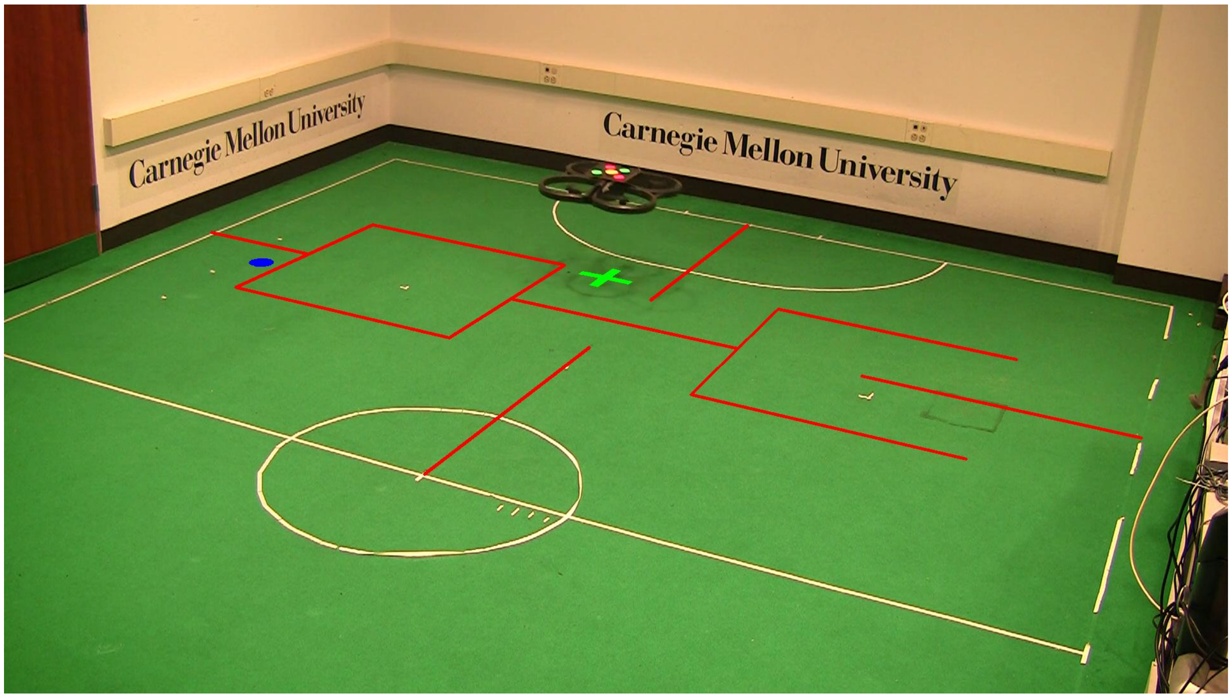


Figure 2.4: A video frame with the obstacles also annotated, showing the virtual environment in which the quadrotor operates.

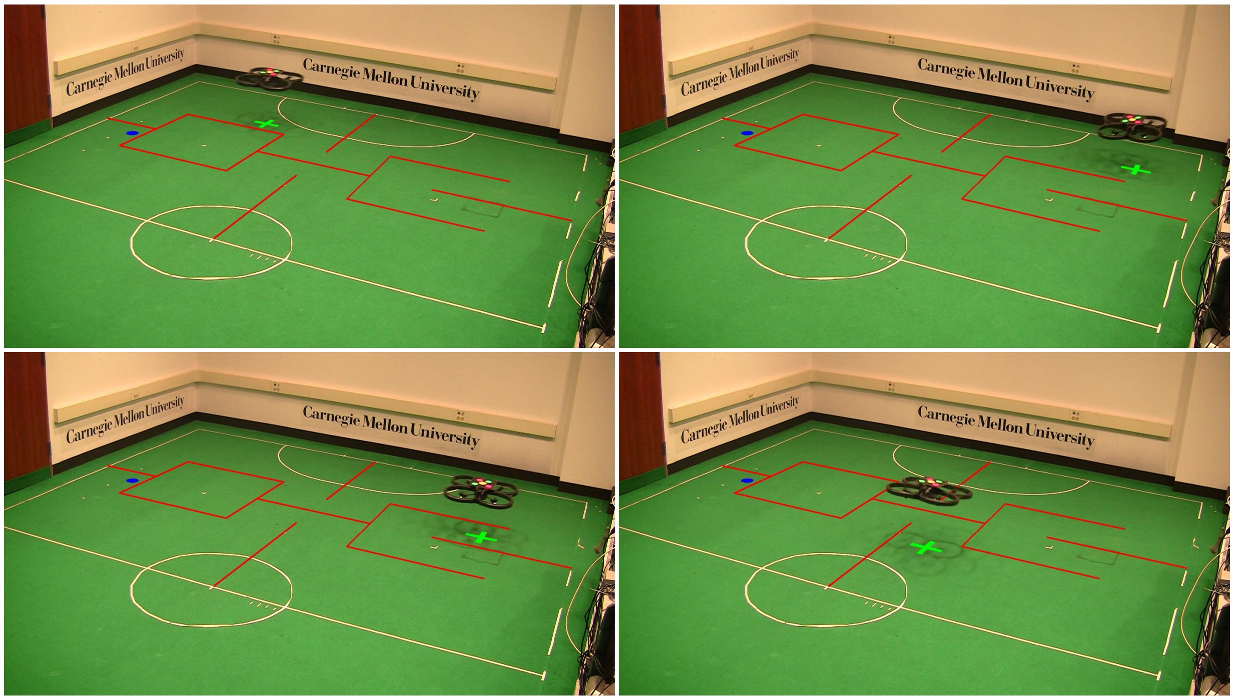


Figure 2.5: A sequence of video frames shown in order, showing how the obstacles can explain the complicated path of the quadrotor as it attempts to reach the blue circle.

time the quadrotor transmits an altitude value, which it does at 15 Hz, the control algorithm uses that value in conjunction with the latest SSL-Vision reading to compute a more accurate position for the quadrotor and runs the RRT algorithm to find a path from that point to the target point. The algorithm computes and sends an acceleration command for the quadrotor based on the path, then waits for the next altitude reading.

The RRT algorithm At a high level, the RRT algorithm performs a search through the state space it is given in order to find a continuous path from the start state to the end state. As we will discuss below, the state space we use corresponds to a region on the ground, so we can think of the algorithm as performing a search directly within the space on the ground.

The simplest state space for an RRT consists of the configuration space of the robot. More complex versions can take into account the dynamics of the robot to compute the connections between states. Using only the configuration space can be a great simplification compared to the true problem, but it is straightforward in concept and implementation, and often leads to good results. For simplicity's sake, we ignore the orientation and altitude of the quadrotor and take the configuration of the robot to include only the point on the ground directly below the center of the quadrotor.

We perform extension by moving a fixed distance in a straight line starting at the start state and going in the direction of the end state. As is typical for kinematic RRTs, the random state generation simply chooses a location uniformly at random within the set of configurations, and the metric is Euclidean distance between locations.

In addition to the state space itself, the RRT algorithm also requires functions to perform the following operations on elements of the space: (a) measure the distance between two elements, (b) generate random elements within the space, (c) check whether two nearby elements can be connected directly without hitting any obstacles, and (d) describe how to start at an element of the space and move a short distance (*extend*) toward any other state. Algorithm 1 describes the basic form of the algorithm. In short, the algorithm explores the state space by repeatedly generating a random element within the space and extending the nearest previously found state toward the new state, if doing so does not run into any obstacles. When the extended state is sufficiently close to a goal state, the algorithm terminates. The entire path is then constructed, as a sequence of states, by backtracking to the start state.

Visualizing the RRT The nature of the RRT as a search algorithm dealing heavily with locations on the ground means that it lends itself naturally to visualization on top of a video of the world. In Line 8 of Algorithm 1, the algorithm creates *new-state* by extending an existing state, *closest-state*, toward a new randomly-generated state, *random-state*. If the direct path from *closest-state* to *new-state* does not hit any obstacles, then *new-state* is recorded as a known state, with *closest-state* as its parent. The structure of the parents of the known states creates a tree of states rooted at the initial state (i.e., the quadrotor's current location). Since, in our application, the elements in the tree are all locations, we can naturally visualize the structure of the tree directly on the ground. For each state s within the tree, we simply draw a line segment between s and its parent *parents*[s], as shown in Figure 2.7.

Making this drawing for each state immediately visually demonstrates how the algorithm

Inputs:

- *start-state*: the element of the state space at which to begin the search (usually the current robot state)
- *goal-state*: the element of the state space to be found by the search

Outputs:

- *path*: a list of states on a path from *start-state* to *goal-state*

Constants and helper functions:

- *distance-threshold*: the distance threshold for counting a state as close enough to the goal
- Dist(u, v): the distance between u and v in the state space
- RandomState(): a randomly-chosen element of the state space
- Extend(u, v): the result of starting at u in the state space and extending toward v
- IsClear(u, v): whether it is possible to directly reach state v from state u

```
 1: function RRT(start-state, goal-state)
 2:     known-states ← {start-state}
 3:     new-state ← start-state
 4:     parents ← empty mapping object
 5:     while Dist(new-state, goal-state) > distance-threshold do
 6:         random-state ← RandomState()
 7:         closest-state ← $\arg\min_{k \in known\text{-}states}$ Dist(k, random-state)
 8:         new-state ← Extend(closest-state, random-state)
 9:         if IsClear(closest-state, new-state) then
10:             known-states ← known-states ∪ {new-state}
11:             parents[new-state] ← closest-state          ▹ set an element in the mapping
12:         end if
13:     end while
14:     path ← [new-state]
15:     while new-state ≠ start-state do
16:         new-state ← parents[new-state]
17:         path ← new-state :: path                        ▹ prepend new-state to path
18:     end while
19:     return path
20: end function
```

Algorithm 1: The basic RRT algorithm.

searches throughout the space, starting from the initial state and branching out toward the goal. But there is even more information we could visualize: we could show each *random-state*, perhaps connected to its corresponding *closest-state*, or include the instances of *new-state* that the algorithm does not add to the tree because the path from *closest-state* goes through an obstacle. As we will discuss more later, the choice we make to not show such information touches on the issue of selection of visualizations; because the other information is, in a sense, less relevant to the output of the RRT, we can omit it without losing as much insight about what the robot does.

These drawings correspond more generally to the idea of moving past mere state and environment information and delving into the planning aspects of a robot's execution. In adding the drawings, we see our first image showing information on something that could be considered truly planning- or intelligence-based. We must make sure to be able to capture such information if we are to adequately express the reasoning of autonomous robot systems. The ability to record and show this planning information is the key reason for working with autonomous robots. When we have access to the computational processes driving the planning and execution of these physical entities, we can make use of it to add quantitative drawings to the videos of execution. If we had, say, videos of humans performing tasks, we would not be able to add additional information that truly described information about the internal processes behind the contents of the video.

2.2.6 Path visualization

Figure 2.7 gives a lot of information about the internals of the computation, but keep in mind that, in a video, the whole tree will be recomputed and redrawn at 15 Hz; due to the randomized nature of the RRT algorithm, the tree will be distinct each time. In our judgment, showing the full set of points in the tree at every frame would be too much information; with many visualizations in a fixed frame area, visual clutter becomes an important issue to avoid. That leads us to our next question:

What is the most relevant information from the robot's execution?

With the issue of visual clutter in mind, we might want to remove the full set of points and only draw, say, the final path to the target point, as those points are the subset of the tree most directly relevant to the final behavior; Figure 2.8 shows the result. With only the final path drawn, we have selected the most relevant subset of information from the RRT algorithm about the planning of the quadrotor.

This issue of the *selection* of visualizations is one of the key ideas that we explore in this thesis. Robots, when executing search-based algorithms like the RRT, can generate a large number of states that can potentially be visualized. While seeing the full set can be useful (e.g., to ensure that the algorithm achieves proper coverage of the space, or that the goal is actually reachable from the start), we imagine that doing so will not often be necessary in such cases. Even in the absence of a single specific algorithm that generates a lot of information, real-world robot algorithms can easily process enough information across their different components to be too much to show all at once.

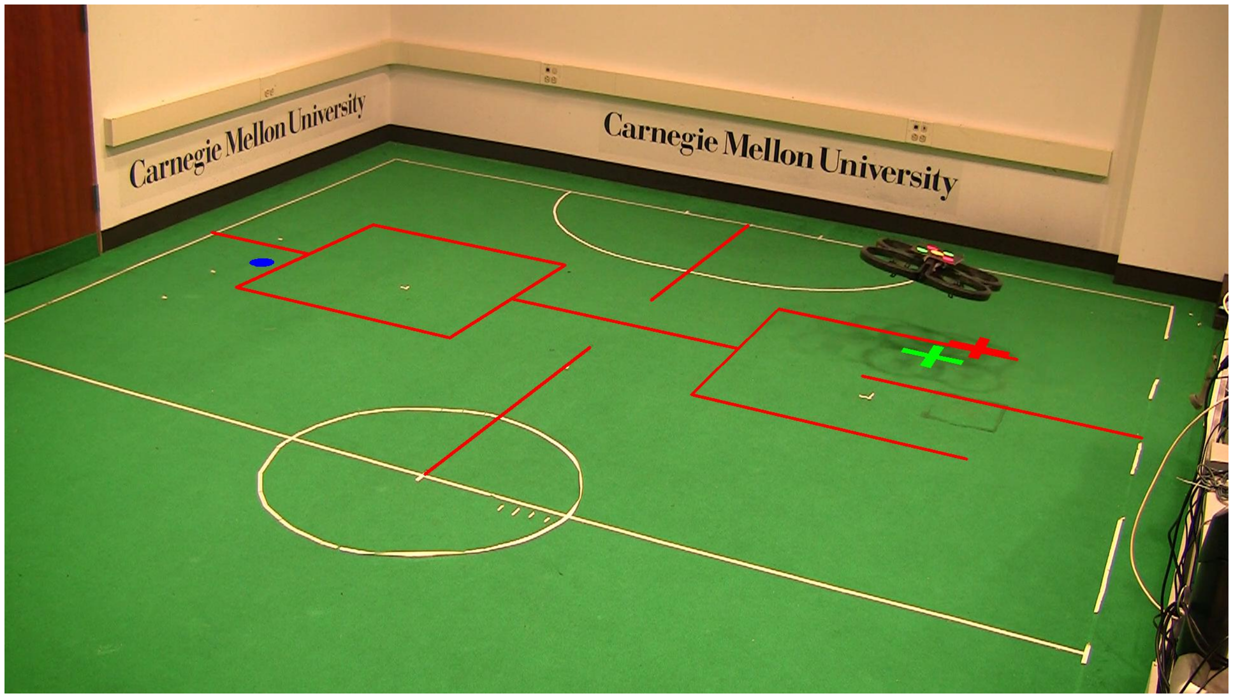


Figure 2.6: A video frame from a different time showing the robot's state after it has illegally crossed an obstacle and is attempting to return to a previous legal location.

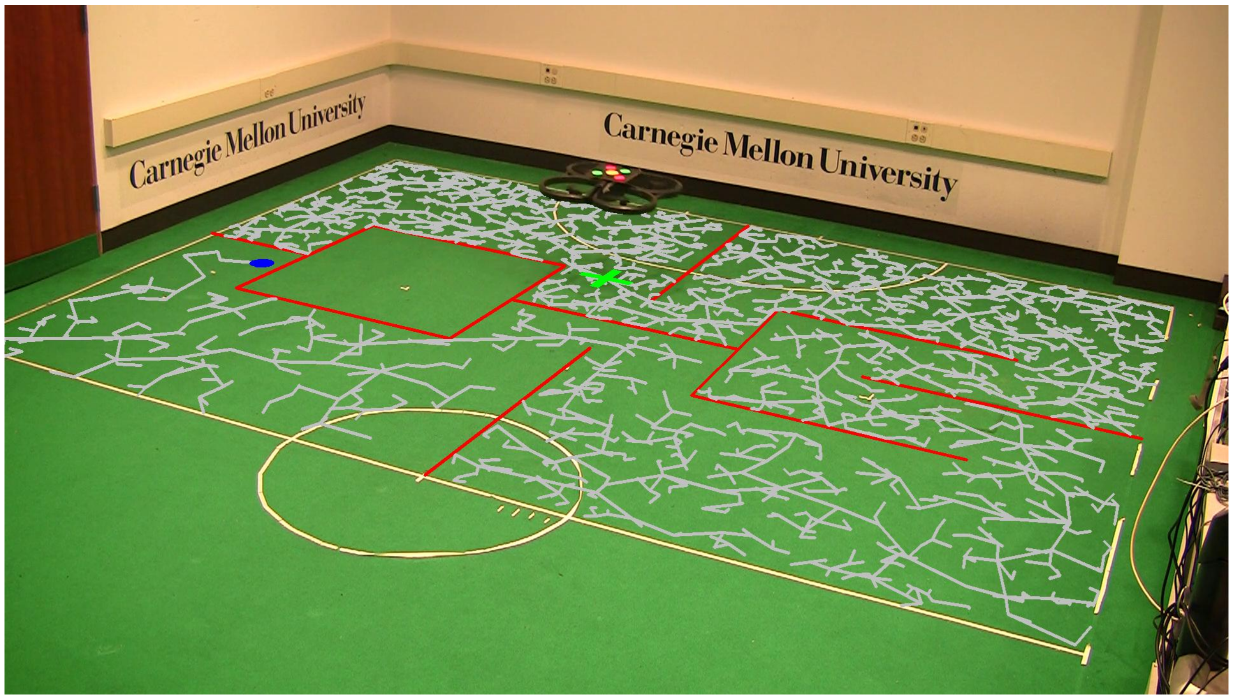


Figure 2.7: A video frame showing the full tree explored by the RRT, visualized as line segments joining each state within the tree to its parent.

2.2.7 Simplified path visualization

A common optimization used with RRTs is to simplify the returned path by finding the last state in the path which is directly reachable from the start state [Bruce and Veloso, 2002] and deleting all states in the path before it (excluding the start state). The state so computed is roughly the first "turn" on the path; navigating toward that point after each RRT run gives the robot a consistent direction to move in. Without this process, the randomness of the RRT means that the first step in the path may take the robot in widely varying directions between timesteps. In our quadrotor domain, After each run of the RRT, the quadrotor attempts to fly toward the point resulting from this optimization.

By going from the full tree to the path, as described in the previous section, we have already selected a small subset of the total available information to display. However, we can further refine what is present as we consider this question:

What information contributes to the motion of the robot?

by depicting the result of path simplification, since the first point after the start on the simplified path is, ultimately, the only value from the RRT that influences the output. Figure 2.9 shows the result of replacing the full path with the simplified path and also marking the first point on the path with a circle. By doing so, we further reduce the amount of information shown, concentrating on what is most relevant to the actions of the quadrotor.

2.2.8 Additional motion information visualization

We still have yet to show the final phase of control: computation of the actual acceleration command to send to the quadrotor. We have gone over the planning information as far as choosing where the quadrotor should try to move, but the results of that planning need to be translated back into commands whose values can directly affect the actual motion of the quadrotor, which leads us to the next question:

How does the robot choose to move given the results of its planning?

The quadrotor accepts pitch and roll commands, which control the horizontal acceleration of the quadrotor, as well as yaw and vertical speed commands, which we use only to hold the yaw and altitude to fixed values. Since acceleration can only be changed by rotating the whole body of the quadrotor, there is latency in the physical response, compounded by the latency in the vision system and communication with the quadrotor.

Thus, we need to supply acceleration commands that allow the quadrotor to move smoothly to a desired point in the face of latency and unmodeled dynamics. Since the low-level control was not the focus of this work, we devised the *ad hoc* algorithm shown in Algorithm 2. It takes the displacement to the target from a projected future position of the quadrotor, computes a desired velocity which is in the direction of the displacement, and sets the acceleration to attempt to match that velocity. The perpendicular component of the velocity difference is more heavily weighted, since we consider it more important to get moving in the right direction than at the right speed.

For brevity's sake, we avoid going through every intermediate vector in the computation and simply add on displaying the final acceleration vector. We turn it into a location on the ground

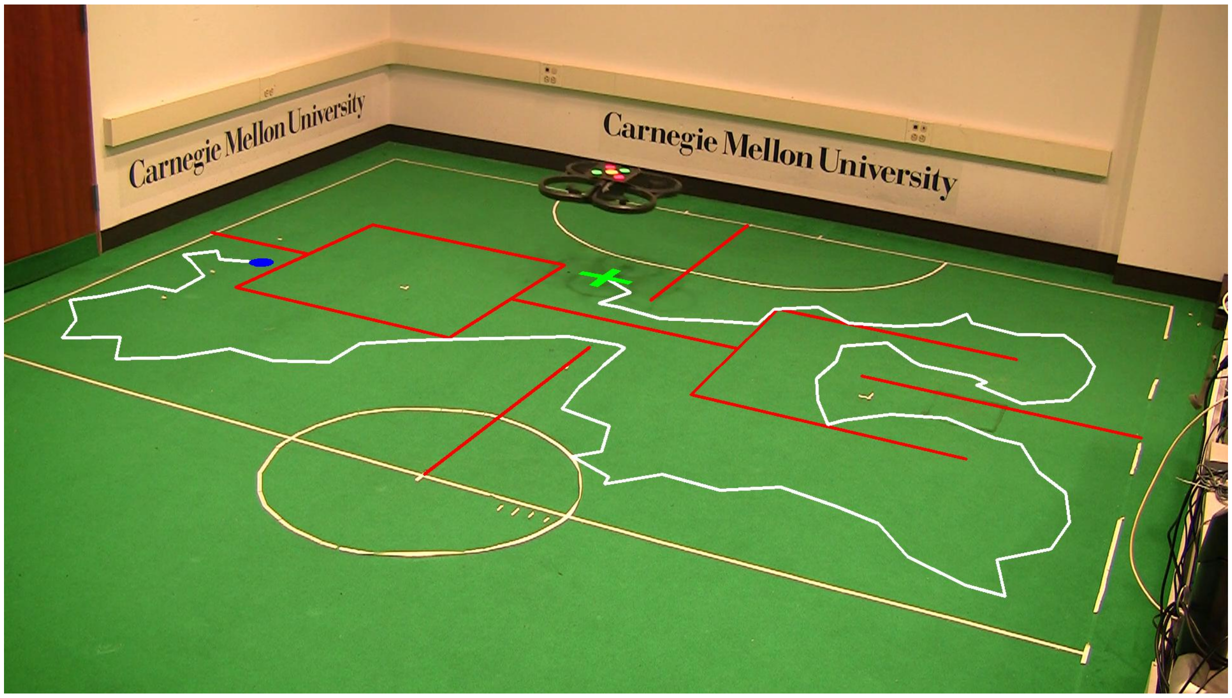


Figure 2.8: A video frame with the full tree removed, leaving only the links between states on the actual final path to the goal.

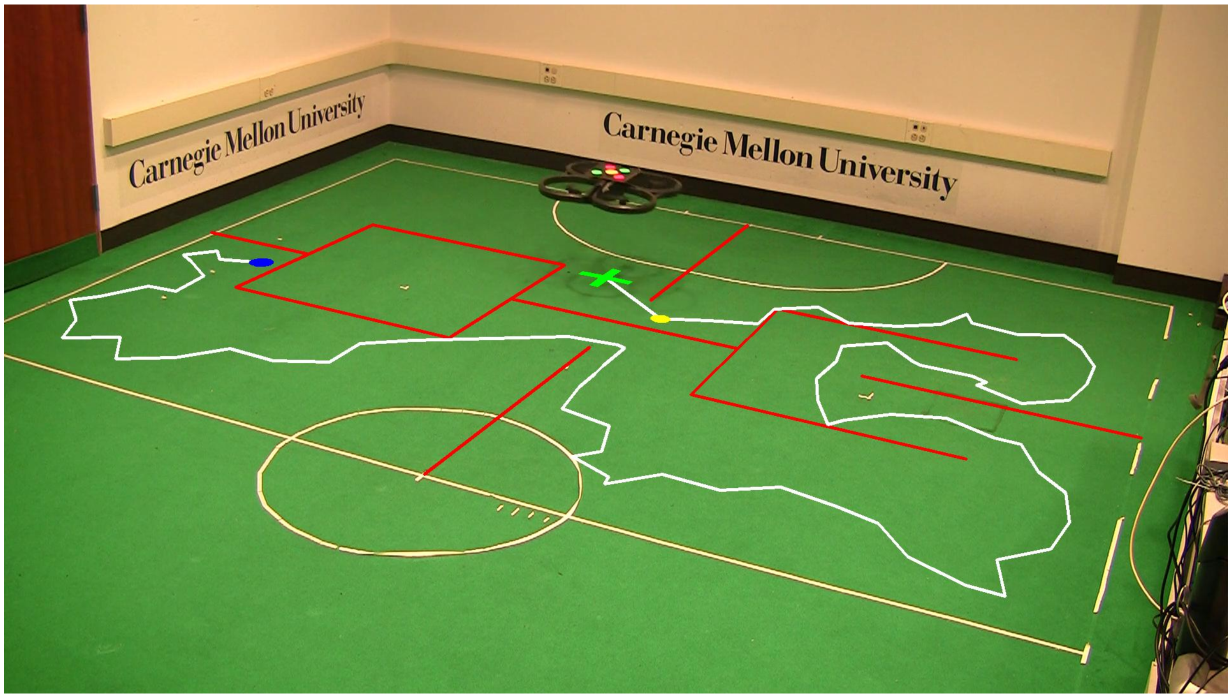


Figure 2.9: A video frame showing the path after simplification and the point to navigate toward.

Inputs:

- *loc*: the current location of the quadrotor
- *vel*: the velocity of the quadrotor, computed by linear regression on the last 5 positions
- *target*: the point to navigate toward, computed by path simplification on the RRT output

Outputs:

- *accel*: the final acceleration vector command sent to the quadrotor

Constants and helper functions:

- $\textsc{Bound}(v, l)$: $\min(l, ||v||)\,\hat{v}$, the vector v with its length bounded to l
- $\textsc{Proj}(u, v)$: $(v \cdot \hat{u})\,\hat{u}$, the projection of the vector v onto u
- $\Delta t_{\text{loc}} = 0.3\,\text{s}$
- $\Delta t_{\text{vel}} = 0.5\,\text{s}$
- $vel_{\text{max}} = 1000\,\text{mm/s}$
- $a_{\text{max}} = 1500\,\text{mm/s}^2$
- $\Delta t^{||}_{acc} = 0.8\,\text{s}$
- $\Delta t^{\perp}_{acc} = 0.5\,\text{s}$

```
1:  function ComputeAcceleration($loc$, $vel$, $target$)
2:      $loc_{\text{fut}} \leftarrow loc + \Delta t_{\text{loc}} \cdot vel$
3:      $\Delta loc \leftarrow target - loc_{\text{fut}}$
4:      $vel_{\text{des}} \leftarrow \textsc{Bound}\left(\frac{\Delta loc}{\Delta t_{\text{vel}}}, vel_{\text{max}}\right)$
5:      $\Delta vel \leftarrow vel - vel_{\text{des}}$
6:      $\Delta vel_{||} \leftarrow \textsc{Proj}(vel_{\text{des}}, \Delta vel)$
7:      $\Delta vel_{\perp} \leftarrow \Delta vel - \Delta vel_{||}$
8:      $accel = \textsc{Bound}\left(\frac{\Delta vel_{||}}{\Delta t^{||}_{acc}} + \frac{\Delta vel_{\perp}}{\Delta t^{\perp}_{acc}}, a_{\text{max}}\right)$
9:      return $accel$
10: end function
```

Algorithm 2: The algorithm that computes the commands to send to the quadrotor.

by scaling it by some constant and adding the resulting vector to the quadrotor's location. In making this choice, we are preemptively applying selection of visualizations as discussed before: since the intermediate vectors are used only by the computation of the acceleration command, and are individually rather simple to compute, we choose to skip them.

With any robot, the final physical motion comes from some sort of control values applied to the physical effectors of the robot, and we imagine that robot control algorithms often use a separation like the one we've discussed here, between planning that produces a target position, which we can visualize, and a motion planning stage that computes velocities and the control quantities for the robots.

2.3 Drawing process

Next, we discuss the process of taking the primitives from a log and drawing them convincingly on the video. We need two key pieces of information for each frame of the video: the set of pixels in the frame that should be drawn on, and the transformation from the coordinates used by the algorithm into video pixel coordinates.

2.3.1 Masking

For the domains where we have implemented our system so far, the ground plane is an SSL playing field, and we want to draw only on the field surface, not on top of the robots or any obstacles on the field. In order to do so, we need to detect which pixels in the video are actually part of the field in each frame. Since the majority of the field is solid green, a chroma keying process (masking based on the color of each pixel) mostly suffices.

We begin by converting the frame to the hue/saturation/value (HSV) color space. which separates hue information into a single channel, making it more robust to lighting intensity changes and well-suited for color masking of this sort. We simply took all the pixels with hue values within a certain fixed range to be green field pixels, providing an initial estimate of the mask.

To account for the field markings, which are not green, we applied morphological transforms [Serra, 1983] to the mask of green pixels. The idea is that the markings form long, thin holes in the green mask, and there are no other such holes in our setup; therefore, a dilation followed by an erosion with the same structuring element (also known as a closing) fills in the field markings without covering any other pixels. We also applied a small erosion beforehand to remove false positives from the green chroma keying. The structuring elements for all the operations were chosen to be iterations of the cross-shaped 3×3 structuring element, with the number of iterations chosen by hand.

2.3.2 Coordinate transformation

Since we are only concerned with a plane in world space, we need a homography to give the desired coordinate transformation, if we assume an idealized pinhole camera [Forsyth and Ponce, 2003]. A pinhole camera projects the point (x, y, z) in the camera's coordinate system to the image coordinates $\left(\frac{x}{z}, \frac{y}{z}\right)$, so the coordinates (u, v) are the image of any point with coordinates proportional to $(u, v, 1)$. Suppose that the ground coordinates $(0, 0)$ correspond to the coordinates $\vec{q}$ in the camera's coordinate system, and that $(1, 0)$ and $(0, 1)$ correspond to $\vec{p}_x$ and $\vec{p}_x$ respectively. Then, for any x and y, the ground coordinates (x, y) correspond to the camera coordinates

$$x\,\vec{p}_x + y\,\vec{p}_y + \vec{q} = \begin{pmatrix} \vec{p}_x & \vec{p}_y & \vec{q} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Thus, multiplying by the matrix $\begin{pmatrix} \vec{p}_x & \vec{p}_y & \vec{q} \end{pmatrix}$ takes ground coordinates to the corresponding image coordinates; the resulting transformation is a homography. There are well-known algorithms to compute the homography that best fits a given set of point or line correspondences.

We used the function for this purpose, `findHomography`, from the OpenCV library [Bradski, 2000].

We primarily used a stationary camera; for such videos, we manually annotated several points based on one frame of the video and used the resulting homography throughout the video. We used intersections of the preexisting SSL field markings as easy-to-find points with known ground coordinates. We also implemented a line-tracking algorithm similar to the one by Hayet et al. [2004], which allows some tracking of the field with a moving camera.

2.3.3 Drawing

Finally, with the above information at hand, drawing the desired shapes is straightforward. To draw a polygon, we transform each vertex individually according to the homography, which gives the vertices of the polygon as it would appear to the camera. Since a homography maps straight lines to straight lines, the polygon with those vertices is in fact the image of the original polygon. Then we fill in the resulting distorted polygon on the video, only changing pixels that are part of the field mask. We used crosses and circles (represented as many-sided polygons) as primitives. Figure 2.11 shows an example of an frame from the output video, along with the image created by drawing the same primitives directly in a 2-D image coordinate system, and Figure 2.12 demonstrates the stages involved in drawing each frame.

2.4 Discussion and summary

In this chapter, we introduced the robotic domain, autonomous navigation by a quadrotor, that led us to exploring the ideas of this thesis, and delved deeply into explaining how to develop a means of visualizing the internals of the robot. Despite the simplicity of the problem that the robot solves here, we can already begin to see some of the questions that drive us to explore more general robot domains.

The presence of virtual obstacles that cause the quadrotor to take complicated paths through apparently open space highlights the importance of bridging the gap between what can be seen in the world, whether by an eye or by a video camera, and what is present and can be shown in the reasoning of the robot. When there are factors influencing a robot's behavior that are not visible in the real world, we wish to make it possible to visually display the effect that they have on that behavior.

The issue of choosing between showing the full RRT search tree, the full path to the goal, and the simplified path leads to the general principle of visualization selection and showing different levels of detail for the same robot execution. A video frame can only hold so much graphical information before becoming too cluttered to be useful, so we need to have a structured way to select from the information produced by a robot algorithm. This selection is an important part of what we explore further later on.

We also showed how to express the choice of visualization as mapping from distinct, meaningful quantities within the robot algorithm to particular kinds of graphical drawings based on the values of the quantities at each timestep. To be explicit, we ultimately depict the quantities from the computation of the quadrotor as shown in Table 2.1. In general, the developers of any

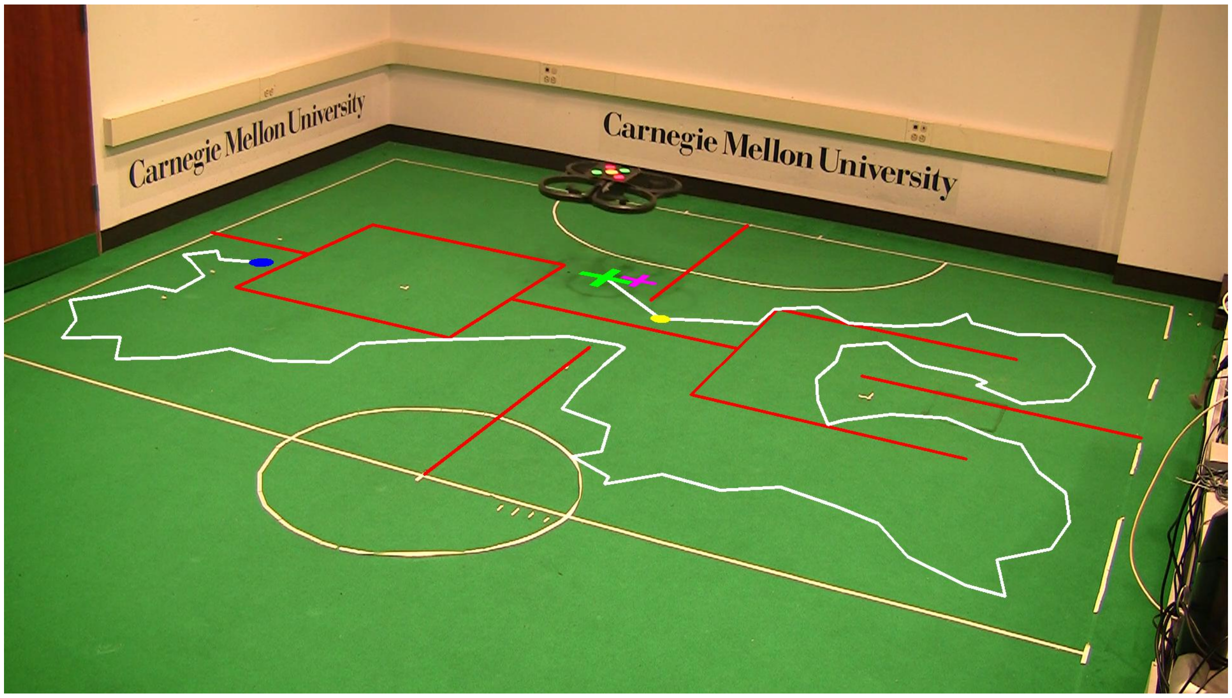


Figure 2.10: A video frame showing the acceleration command sent to the robot.

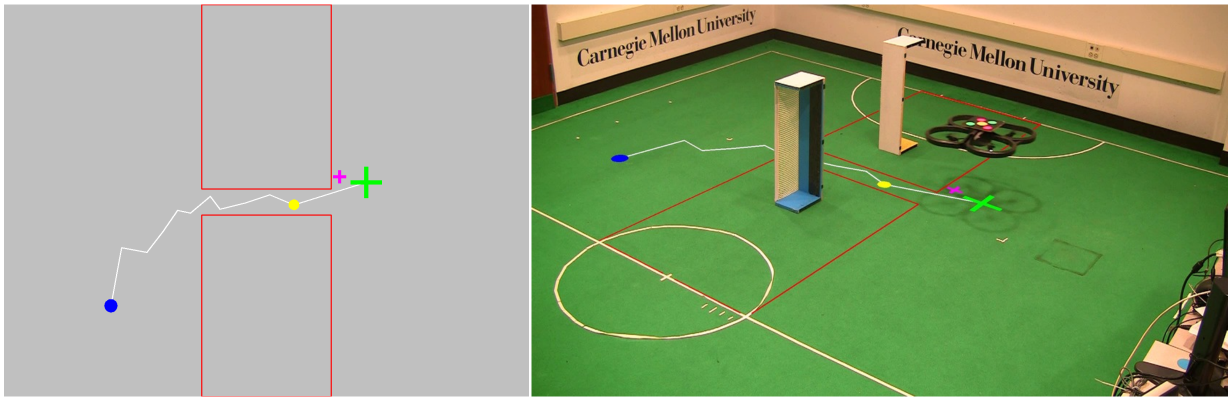


Figure 2.11: Left: A 2-D visualization generated from one run of the RRT, using a domain based on physical obstacles. Right: The result of drawing that visualization onto the corresponding frame from the video of the real robot.

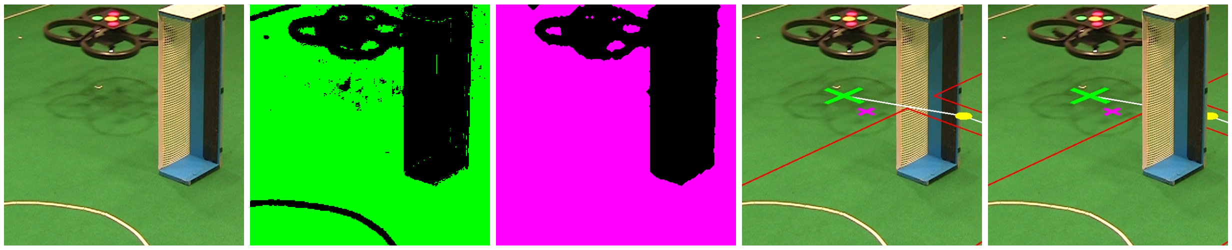


Figure 2.12: The stages in computing the mask for a frame of a video to properly overlay our visualizations. From left to right: (1) the original frame, (2) the raw green mask, (3) the mask, after morphological operations, (4) the result of drawing without taking the mask into account, and (5) the drawing, masked appropriately.

Quantity	Drawing
current location of the quadrotor	✚ green cross
current target point	● blue circle
final acceleration command	+ pink cross
virtual obstacles in the environment	╱ red lines
links between points searched by RRT	╱ gray lines
final computed path to the target	╱ white lines
first point on simplified path	• yellow circle

Table 2.1: The mappings from quantities computed by the quadrotor to drawings as shown in the figures of this chapter.

individual robot may choose a mapping and set of quantities to suit their particular algorithm and application. Within this thesis, we will not address the problem of selecting quantities and mappings given an arbitrary robot algorithm; we will treat those choices as given and focus on how to display the results of the choices given algorithm output and a video.

A further question that we have not addressed is: what else is there to the operation of the quadrotor that we have *not* visualized? There are further concepts that are part of the overall execution but that we have not discussed. For example, could we try to somehow depict the transmission of commands to the quadrotor over Wi-Fi, or the spinning of the rotors? Those concepts are more distant from the core principle that we focus on: the physical motion of the position of the robot through space. Accordingly, we leave out less relevant concepts and stick to working with quantities that are more directly related to the spatial behavior of the quadrotor behavior: those involving positions or velocities within the space.

Chapter 3

Domain-independent visualization

Now that we have established a motivating example and shown how we can use visualizations with an autonomous robot, we move on to setting up a means to create such drawings for any robot, or even any context at all. In Section 3.1, we begin by describing our methods for organizing any set of geometric drawings that change over time, such as those that might be generated by an autonomous robot's reasoning. In Section 3.2 we discuss concrete details of our software implementation of the methods of this chapter. Finally, in Section 3.4, we return to see how the principles outlined in the previous sections apply to the quadrotor example domain specifically.

3.1 Robot-independent visualization language

We have described the motivating example of the quadrotor domain, but, of course, we wish to create something that can apply to any robot system. We contribute a description of a structure for encoding a set of drawings that vary across time in a platform-neutral format to allow for later viewing; we describe that format and its usage in this chapter.

3.1.1 Language description

We describe the language using the grammar in Figure 3.1, in extended Backus-Naur form. (We describe a human-readable grammar for descriptive purposes here; for concrete implementation on a computer, we use an equivalent binary representation, as described in Section 2.3.)

A simple example of a log conforming to this grammar is in Figure 3.2 (with extra spacing for legibility). A real-world example is in Figure 3.7; its entries correspond to the drawings in Figure 2.11.

Informally, a log consists of a set of *frames*; each frame has a *time specification*, consisting of one or two timestamps (the meaning of which is discussed in Section 3.1.2), as well as a list of *primitives* (or *drawing primitives*). Each primitive contains both a description of a geometric shape and information related to filtering. The shape description of each primitive is one of a few types of parametrized geometric shapes. The above grammar describes circles, lines, and rectangles, which we use in our implementation, but other shapes (e.g., arbitrary polygons) could

$$
\begin{array}{rcl}
\langle\text{log}\rangle & \models & \{\ \langle\text{frame}\rangle^{*}\ \} \\
\langle\text{frame}\rangle & \models & \{\ \langle\text{time-specification}\rangle\ ;\ \langle\text{primitives}\rangle\ \} \\
\langle\text{time-specification}\rangle & \models & \textit{time}\ \mid\ \{\ \textit{start-time}\ ;\ \textit{end-time}\ \} \\
\langle\text{primitives}\rangle & \models & \{\ \langle\text{primitive}\rangle^{*}\ \} \\
\langle\text{primitive}\rangle & \models & \{\ \langle\text{geometry}\rangle\ ;\ \textit{color}\ ;\ \langle\text{filter-info}\rangle\ \} \\
\langle\text{geometry}\rangle & \models & \langle\text{circle}\rangle\ \mid\ \langle\text{line}\rangle\ \mid\ \langle\text{rectangle}\rangle\ \mid\ \langle\text{polygon}\rangle \\
\langle\text{circle}\rangle & \models & \{\ \text{circle}\ ;\ \textit{center}\ ;\ \textit{radius}\ \} \\
\langle\text{line}\rangle & \models & \{\ \text{line}\ ;\ \textit{start}\ ;\ \textit{end}\ \} \\
\langle\text{polygon}\rangle & \models & \{\ \text{polygon}\ ;\ \textit{vertex}^{*}\ \} \\
\langle\text{rectangle}\rangle & \models & \{\ \text{rectangle}\ ;\ \textit{center}\ ;\ \textit{size}\ ;\ \textit{orientation}\ \} \\
\langle\text{filter-info}\rangle & \models & \textit{entity}\ ;\ \textit{level}
\end{array}
$$

Figure 3.1: A grammar describing the contents of the visualization logs. Asterisks indicate zero or more repetitions of the preceding values; angle brackets contain nonterminal names; italics indicate descriptive text; pipe characters indicate alternatives; other characters are literals.

be added without substantially altering any of the following discussion. The primitives within a frame also contain, as shown by the *filter-info* element, additional information about their meaning within the context of the log, in order to allow for meaningful selection of different subsets of the information. In Section 3.1.4, we further discuss this filtering.

The timestamps may be specified in any suitable format, such as seconds since the Unix epoch. The geometric elements (center, radius, start, end, size, orientation) may be specified in relation to any consistent and known reference frame. We will later discuss combining this language with videos. For some applications, it may suffice to use the video coordinates; typically, coordinates based in the real world are more useful.

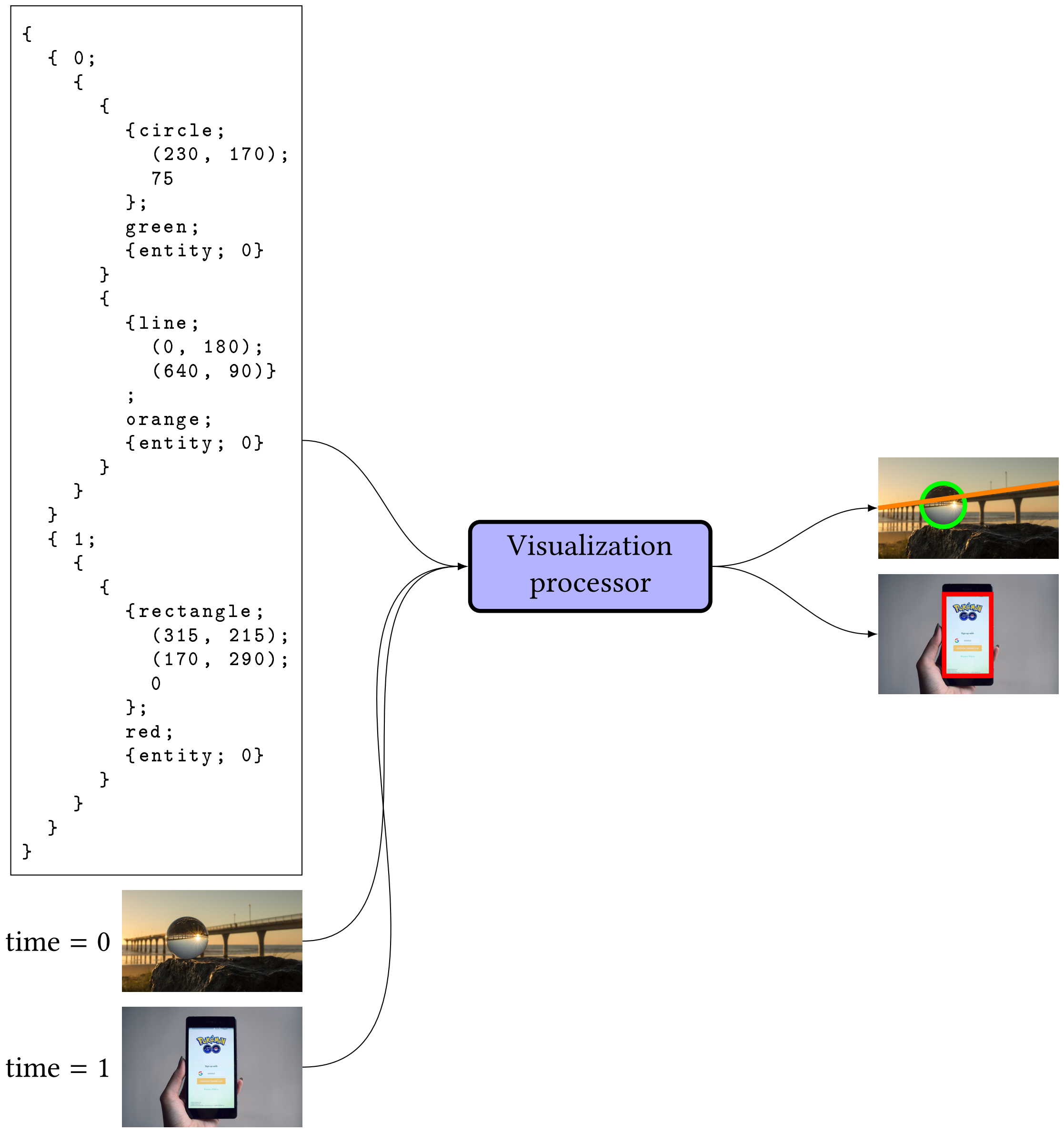
3.1.2 Instantaneous and persistent frames

We divide the frames into two categories: *instantaneous* and *persistent*. Instantaneous frames have a single timestamp associated with them, while persistent frames a start and an end time. When logs are viewed, it is necessary to know what frames are relevant at any particular time; instantaneous and persistent frames differ in how that determination is made.

Instantaneous frames are intended to describe transient information relating to animations or rapidly changing data, such as tracking the position of an object depicted in the video. At most one instantaneous frame is relevant at any given time: the one with the closest timestamp to the query timestamp, if the two timestamps are close enough together.

Persistent frames are useful for longer-lasting information, such as static obstacles. A persistent frame is visible any time the query timestamp is between the start and end times of the frame. There may be multiple such frames visible if their time intervals overlap.

The functionality of instantaneous frames could be essentially replicated using persistent



```
{
  { 0;
    {
      {
        {circle;
          (230, 170);
          75
        };
        green;
        {entity; 0}
      }
      {
        {line;
          (0, 180);
          (640, 90)}
        ;
        orange;
        {entity; 0}
      }
    }
  }
  { 1;
    {
      {
        {rectangle;
          (315, 215);
          (170, 290);
          0
        };
        red;
        {entity; 0}
      }
    }
  }
}
```

Figure 3.2: A conceptual example of applying the visualization language on top of generic images. Here, we simply use an image space coordinate system for drawing on some sample images; see Section 2.3 for other possibilities. Images courtesy of `https://pixabay.com`.

frames with appropriate time intervals, which would seem to simplify the language. However, we view the distinction as a conceptual difference between information that applies to an extended interval of time and information that is derived from one state of the world and can be discarded and recomputed immediately once the world changes.

3.1.3 Drawing primitives

Each drawing primitive in a log describes a simple geometric shape. We use four kinds of shapes: circles, lines, rectangles, and general polygons. Each primitive contains a color and the geometric parameters necessary to fully describe the shape:

- circle: center, radius;
- line: endpoints;
- rectangle: center, dimensions, orientation;
- polygon: list of vertex coordinates.

One may additionally add other visual properties for the drawings, such as color gradients, thickness, or fill; such additions do not materially affect the rest of the process, so we omit any discussion of them.

3.1.4 Visualization filtering/levels of detail

Since there may be many drawing primitives contained in a log, we would like to be able to select meaningful subsets of the information that is available to display. We generalize the idea of *layered disclosure*, which describes a selection process for textual logs based on a single depth parameter. For visual information, filtering is even more important, since visualizations additionally have the problem of overlapping with each other in the display.

To enable such selection, the drawing primitives also contain, apart from their geometric information, information about the origin and significance of the shape within the log. To organize the set of primitives, we assign an *entity*, which is a human-selected string value, to each primitive.

Currently, we use individually chosen values for the entity names; whether each entity is drawn can be toggled on or off individually within the viewer (described further in Section 3.2.1). We need not impose any further structure on the entity names, but we can; e.g., we can also define a hierarchical structure of entities, where each entity occupies one vertex of a directed acyclic graph (DAG). In that case, the additional graph structure means that we can perform more advanced filtering operations on the set of visualization. We can, for example, only display the entities that are descendants of some particular entity in the graph, or only exclude such entities, or select any arbitrary set of entities at will.

A DAG provides a flexible structure that subsumes simpler cases, such as a set of independent entities or a linear ordered list. Using a DAG also allows us describe the dependencies between different pieces of information contained in different entities, which is one natural way of assigning the entities to different parts of the autonomous agent's algorithms. We describe concrete uses of both flat- and graph-structured entities in two robotic domains in Section 3.2.

We also define an orthogonal system of numerical levels of detail, similar to the original idea of layered disclosure. For any given maximum level and set of entities, the viewer only displays drawings that both have sufficiently low levels and are associated with chosen entities. We intend for the entities to represent qualitatively different aspects of the log, while the numerical levels represent quantitative amounts of detail relating to information from particular aspects.

Algorithm 3 describes the algorithm for selecting drawings from a log, given the time of the video frame to draw on and the other drawing parameters. Storing the instantaneous frames in sorted order by time and the persistent frames in an interval tree would allow the selection of frames to take $O(\log(n) + \log(m) + k)$ time, where k is the number of relevant persistent frames, n is the total number of persistent frames, and m is the total number of instantaneous frames.

The drawing process would then be linear in the number of relevant frames and the number of drawings within those frames. During our usage of the viewer program, we have found that the time spent selecting frames is dominated by the time spent actually drawing the graphics; overall, the system runs well enough for smooth user interaction.

3.2 Concrete software implementation

So far, we have described the conceptual framework for our visualizations, without reference to how to practically create and observe them. In this section, we describe the implementations that we have created to concretely demonstrate the functionality of the visualizations.

3.2.1 Viewer software and interface

We implement a means of displaying the information within a log overlaid on the frames of a video; Figure 1.3 shows an instance of the displayed interface. At any given moment, the viewer shows one frame of the input video, along with a subset of primitives from the associated frames; the frames are chosen according to their timestamps, as described in Section 3.1.2. The primitives to be drawn are those according to their associated entities and levels, as described above. The viewer must also provide an interface for manipulating the filtering of visualizations (i.e., choosing entities and a maximum level). For simplicity, we display the set of entities as a flat list and allow each entity to be toggled on or off independently.

Our implementation is written in the Python programming language[1] and uses the wxWidgets library[2] for its graphical interface. Python and the other dependencies of our implementation are readily available on all major operating system platforms, making it easy to use the viewer with existing robots.

3.2.2 Log storage format

In Section 3.1, we used a formal grammar to describe the structure of the information contained within a visualization log; for actual storage and communication, we use Protocol Buffers

[1] https://www.python.org/
[2] https://wxwidgets.org/

Inputs:

- *video-time*: timestamp of the video frame
- *log-level*: minimum log level
- *entities*: set of selected entities
- *instantaneous-frames*: instantaneous frames
- *persistent-frames*: persistent frames

Outputs:

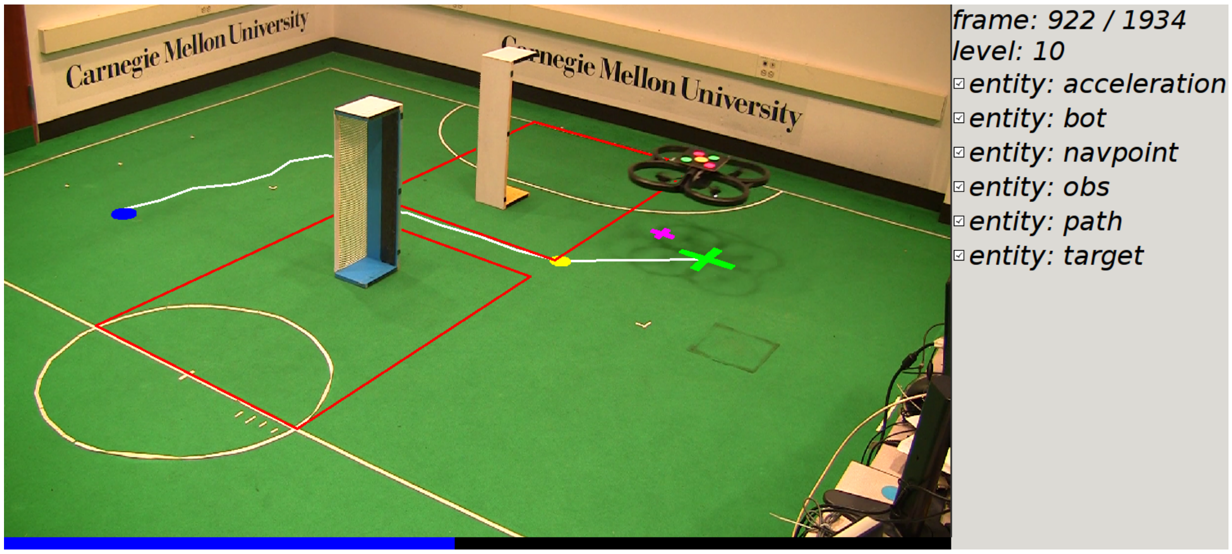
- *drawings*: the final list of drawings to display

Constants and helper functions:

- Time(f), StartTime(f), EndTime(f)
- Drawings(f): the list of drawing primitives contained in a frame
- Entity(d) the symbolic entity of a drawing
- Level(d) the numerical log level of a drawing
- *TimeThreshold*: time difference threshold for instantaneous frames

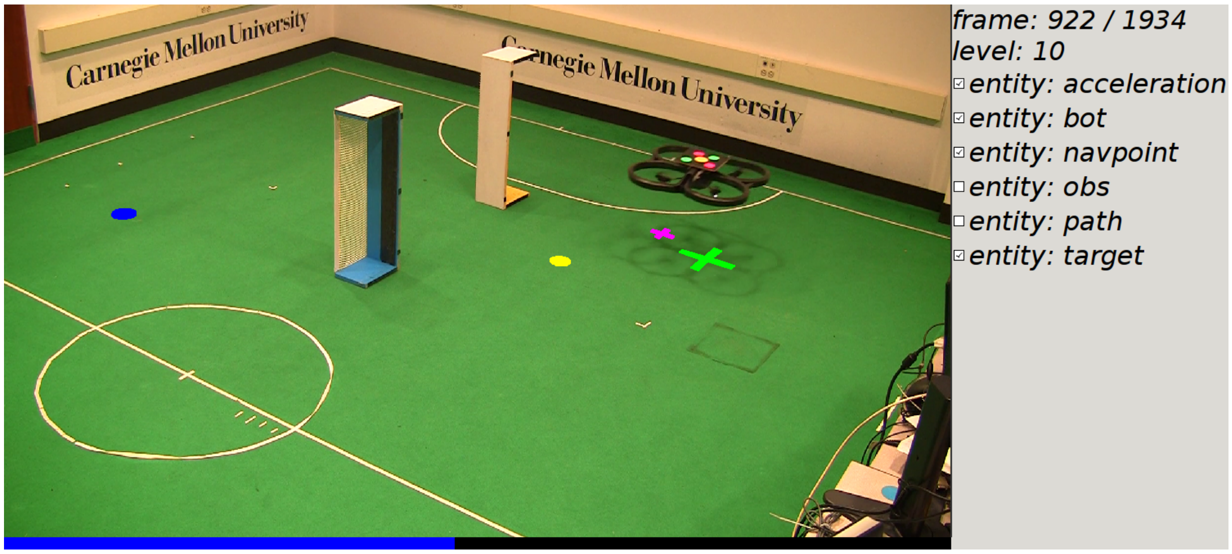
```
1:  function Draw($\textit{video-time}, \textit{log-level}, \textit{entities}, \textit{instantaneous-frames}, \textit{persistent-frames}$)
2:      $\textit{frames} \leftarrow \{f \mid f \in \textit{persistent-frames} \land \textsc{StartTime}(f) \le \textit{video-time} \le \textsc{EndTime}(f)\}$
3:      $\hat{f} \leftarrow \arg\min_{f \in \textit{instantaneous-frames}} |\textsc{Time}(f) - \textit{video-time}|$
4:      if $\left|\textsc{Time}\left(\hat{f}\right) - \textit{video-time}\right| < \textit{TimeThreshold}$ then
5:          $\textit{frames} \leftarrow \textit{frames} \cup \{\hat{f}\}$
6:      end if
7:      $\textit{drawings} \leftarrow []$
8:      for $f \in \textit{frames}$ do
9:          for $d \in \textsc{Drawings}(f)$ do
10:             if $\textsc{Entity}(d) \in \textit{entities} \land \textsc{Level}(d) \le \textit{log-level}$ then
11:                 $\textit{drawings} \leftarrow d :: \textit{drawings}$
12:             end if
13:         end for
14:     end for
15:     return $\textit{drawings}$
16: end function
```

Algorithm 3: The algorithm for selecting drawing primitives from a log to draw onto one frame of a video.



frame: 922 / 1934
level: 10
☑ entity: acceleration
☑ entity: bot
☑ entity: navpoint
☑ entity: obs
☑ entity: path
☑ entity: target

Figure 3.3: The viewer we developed for our cross-robot visualizations, showing the main graphical view, as well as controls for the video and selection of entities.



frame: 922 / 1934
level: 10
☑ entity: acceleration
☑ entity: bot
☑ entity: navpoint
☐ entity: obs
☐ entity: path
☑ entity: target

Figure 3.4: The cross-robot visualization viewer showing the quadrotor, with not all entities selected.

(protobufs)[3], a commonly-used serialization format; the ability to produce log files in the format can be easily incorporated into or added to existing programs or systems. We chose protobufs because of our previous familiarity with them and because they have mature implementations on all common platforms and interfaces with a wide variety of programming languages, making them easy to integrate with virtually any existing robotics platform. In comparison to other widely used platform-neutral serialization formats, such as JSON or XML, protobufs (a) are more compact when stored on disk or sent over the network, (b) are faster to parse and serialize, and (c) provide a simpler and more type-safe interface for programmers[4].

3.3 Details of drawing process

In this section, we extend Section 2.3 by providing additional technical detail on how we carry out the processes of drawing onto an image in general. As before, we break the process down into two components: (a) selecting pixels to draw on and (b) transforming or mapping the coordinates of the log into those of the video.

3.3.1 Masking

Here, we discuss the mechanics of the masking process. We handle image data in the form of multidimensional arrays from the NumPy library[5], along with additional functions from the OpenCV library[6], which operate on NumPy arrays. An image of size $W \times H$ pixels is represented as an array of size $W \times H \times 3$ containing 8-bit red/green/blue (RGB) or HSV values for each pixel; we may also take one of the components as a $W \times H$ array by itself. A mask of the same size is represented as an array of size $W \times H$ containing Boolean values.

The algorithm we use to compute the mask for the SSL field domains relies on estimating three intermediate masks: (a) the visible green pixels, (b) the visible black pixels, and (c) the visible white pixels. Roughly, the process is to take all the green pixels, along with those white pixels that are near green pixels but not black ones. The output and intermediate masks are binary arrays of the same size as the image.

In Algorithm 4, Lines 3–14 estimate the range of hues to allow for green pixels. The algorithm computes the histogram of hue values and finds the most common value, assuming that the background color is the most prevalent in the image. It then searches outward until the histogram reaches a fraction of the peak value, to account for the width of the peak in the histogram. Ultimately, h_0 and h_1 describe the range of hue values we count as green.

Line 15 then selects the pixels with matching hues and Line 16 performs some morphological operations to remove stray pixels and widen the mask slightly. The resulting mask is nearly complete, but we would like to include the lines without covering the quadrotor.

Selecting the lines by pure pixelwise filtering is complicated by the fact that the highlights on the quadrotor body can have similar values to the line pixels. Accordingly, we find a mask of

[3]`https://developers.google.com/protocol-buffers/`
[4]`https://developers.google.com/protocol-buffers/docs/overview`
[5]`http://numpy.org`
[6]`https://opencv.org`

white pixels and a mask of black pixels, then ignore the white pixels that are sufficiently near black pixels.

Line 17 finds a mask of pixels that are dark without being green, which we take to represent the dark body of the quadrotor. Line 18 expands this mask, providing the definition of "near" from the previous paragraph.

Line 19 selects pixels that are white but not near black pixels and Line 20 expands the mask enough that it can overlap with the green mask. Then Line 22 computes a modified green version of the mask that, roughly, fills in sufficiently small holes within it without expanding its outside boundary. Line 23 find the overlap of the white mask with the modified green mask to find only the white pixels within the field, ignoring walls and the like.

Finally, Line 24 computes the final mask as the combination of green pixels and white pixels.

Once we have computed the mask, the final step, that of actually applying it to an image, relies on another operation, which takes three arrays of the same size as arguments: CopyTo(*dest*, *source*, *mask*) (named for NumPy's `copyto` function[7], which we use in some cases). *source* and *dest* are arrays representing images; *mask* is a mask as described above (a Boolean array). The value of CopyTo(*dest*, *source*, *mask*) is an array with the same size as *source* and *dest* is to modify the array *dest* so that for each location where *mask* has a *True* value, the value of *dest* in that location is replaced by the value of *source* at that location.

When we call the function, we supply the result of drawing the primitives onto a frame as *dest* and the original unannotated frame as *source*, while *mask* is a mask array, computed as described above, that has *True* values in the foreground pixels and *False* in the background. The result of CopyTo is an array that is equal to *source* in the locations where *mask* is *True* and *dest* where *mask* is *False*, which achieves exactly the desired effect of drawing only on the background pixels.

3.3.2 Coordinate transformation

For our applications, we map drawings from a single plane in the world into our images. To compute doing so, we require at least four point correspondences between the image coordinates and the world coordinates; we can use more to provide robustness against error in the mappings. There are well-known, robust algorithms for computing the mapping from a set of correspondences [Hartley and Zisserman, 2003]; we use the `findHomography` function of OpenCV, which minimizes the *transfer error* (the sum of squared distances between each image point and the projected location of the corresponding world point)[8]. In Figure 3.6, we visually demonstrate the process of creating and applying the homography mapping between the two coordinate systems.

The use of a homography does assume the use of an idealized camera and lens with no distortion (i.e., straight lines in the world map to straight lines in the video). While some classes of lenses are prone to distortion (e.g., wide-angle lenses) or intentionally distorted (e.g., fisheye lenses), distortion is generally not desirable; the videos we have produced, on a consumer video camera, have no noticeable distortion, as is typical for modern devices. In any case, it is

[7]`https://docs.scipy.org/doc/numpy/reference/generated/numpy.copyto.html`

[8]`https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html`

Inputs:

- *frame*: the image frame to operate on

Outputs:

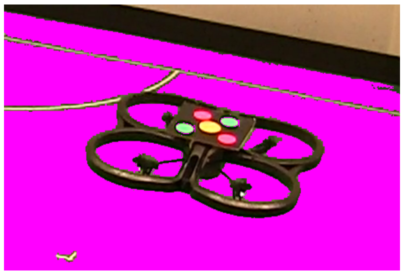
- *mask*: the output binary mask, with *True* values where the pixels show the background

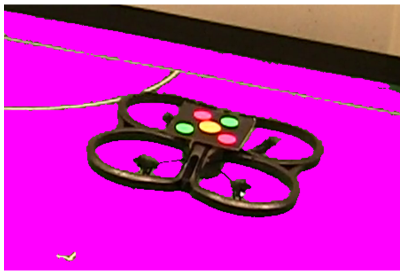
Constants and helper functions:

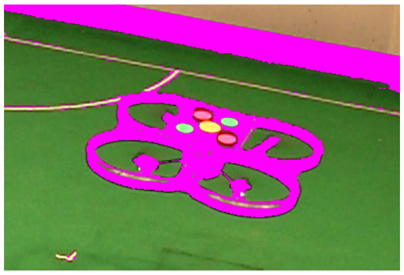
- DILATE(a, n): binary dilation of Boolean array a using a 4-connected structuring element n times
- ERODE(a, n): the binary erosion of Boolean array a using a 4-connected structuring element n times
- HSV(a): the conversion of image a from RGB to HSV color space, expressed as a tuple containing the hue, saturation, and value components as individual arrays
- MORPH(a, $[n_0, \ldots, n_k]$): the result of applying a morphological operation for each n_i to array a in order; if $n_i > 0$, the function applies a binary dilation of n_i iterations, and if $n_i < 0$, it applies a binary erosion of $-n_i$ iterations

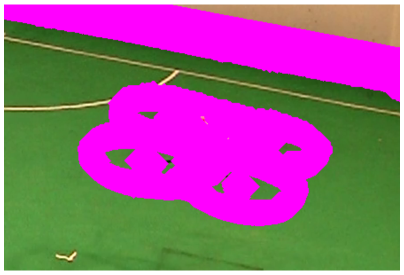
```
function ComputeMask(frame)
    $h, s, v \leftarrow$ HSV(frame)
    $histo \leftarrow$ Histogram($h$)
    $\hat{h} \leftarrow \arg\max_x histo[x]$
    $h_0 \leftarrow \hat{h}$
    $h_1 \leftarrow \hat{h}$
    while $h_0 > \hat{h} - 10 \wedge histo[h_0] > .05 \cdot histo[\hat{h}]$ do
        $h_0 \leftarrow h_0 - 1$
    end while
    while $h_1 < \hat{h} + 10 \wedge histo[h_0] > .05 \cdot histo[\hat{h}]$ do
        $h_1 \leftarrow h_1 + 1$
    end while
    $h_0 \leftarrow h_0 - 12$
    $h_1 \leftarrow h_1 + 12$
    $green \leftarrow (h_0 < h) \wedge (h < h_1) \wedge (s > 100) \wedge (v > 70) \wedge (v < 190)$
    $green \leftarrow$ Morph($green$, $[-1, 2]$)
    $black \leftarrow (v < 120) \wedge \neg green$
    $black \leftarrow$ Morph($black$, $[-1, 8]$)
    $white \leftarrow (v > 130) \wedge (s < 120)$
    $white \leftarrow white \wedge \neg black$
    $white \leftarrow$ Morph($white$, $[4]$)
    $green\text{-}fill \leftarrow$ Morph($green$, $[-1, 14, -13]$)
    $lines \leftarrow white \wedge green\text{-}fill$
    $mask \leftarrow green \vee lines$
    return $mask$
end function
```

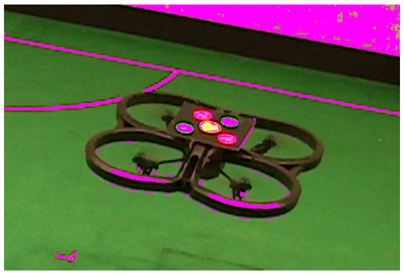
Algorithm 4: The algorithm for computing a background mask based on an image frame. We use ∧, ∨, and ¬ to denote element-wise conjunction, disjunction, and negation of Boolean arrays. Also, < and > show element-wise comparison between one array and one scalar, evaluating to a Boolean array. Figure 3.5 demonstrates the values of the masks for an example image.

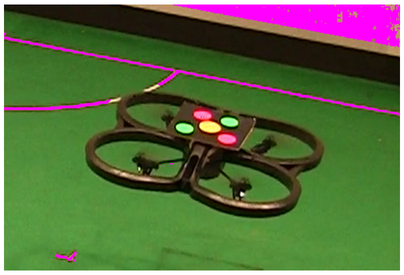

(a) *green* (Line 15)

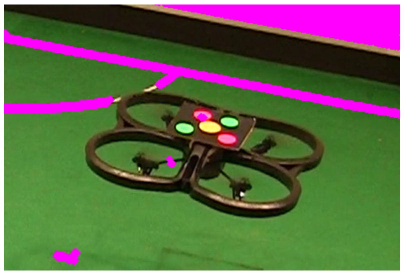

(b) *green* (Line 16)

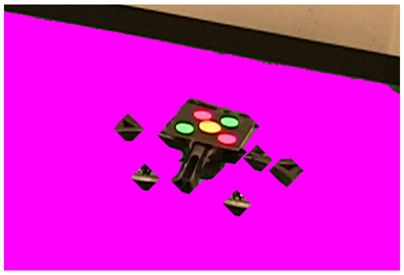

(c) *black* (Line 17)

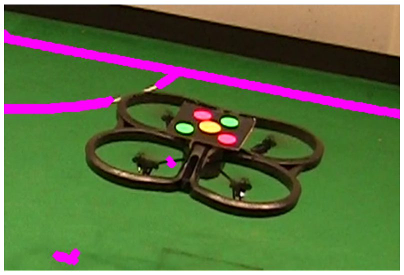

(d) *black* (Line 18)


(e) *white* (Line 19)


(f) *white* (Line 20)


(g) *white* (Line 21)


(h) *green-fill* (Line 22)


(i) *lines* (Line 23)

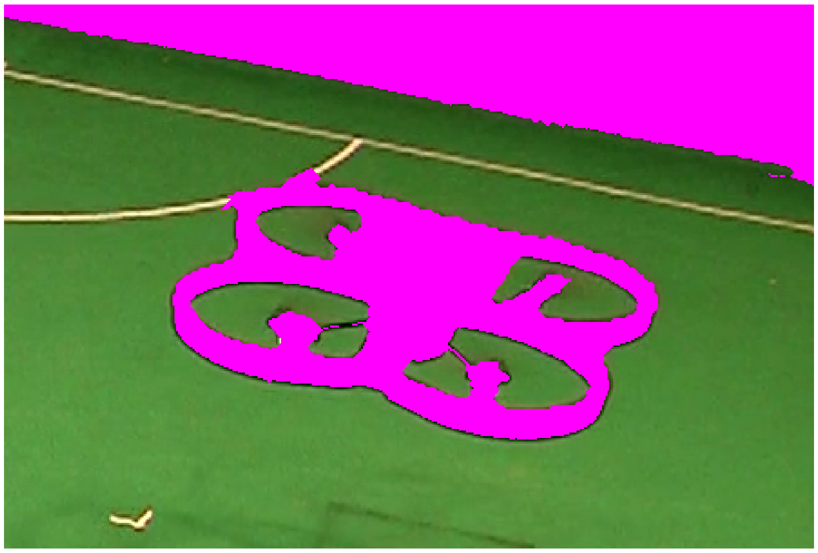

(j) *mask*

Figure 3.5: A demonstration of the stages in the process of masking. In each image, we draw the pixels present in the mask as pink on top of the original image for visibility; the masks are actually stored using a single binary value per pixel.

straightforward to compute and reverse the distortion of any camera setup, given the ability to take new images with it [Forsyth and Ponce, 2003], so we do not consider camera distortion to be a major issue.

Next, we discuss the transformation of an element of the log, which gives a bare description of a geometric shape in world coordinates, into concrete drawings in the image space; Algorithm 5 shows the processes involved. We need to take a specification of the form, e.g., “circle at location (x, y) with radius r” into drawings; homographies can map circles to ellipses, so we cannot simply draw a circle using the underlying drawing library.

However, homographies do map straight lines to straight lines, so we can draw polygons simply by mapping their vertices individually and drawing segments between the resulting points.

While it is possible to analytically determine the form that a circle would map to under a given homography — it is always a conic section [Hartley and Zisserman, 2003] — we choose a conceptually (and implementation-wise) more straightforward sampling-based method: we sample points evenly spaced around the circle and map each point individually using the homography, treating the circle as equivalent to a many-sided polygon. Algorithm 5 demonstrates the drawing process for the different shapes. There, we make use of an abstract DrawImageLine function; in practice, we use some preexisting drawing library, depending on the context. For offline drawing, we use the drawing functions of OpenCV; in the interactive viewer, we use the drawing functions of wxWidgets.

3.3.3 Separation of components

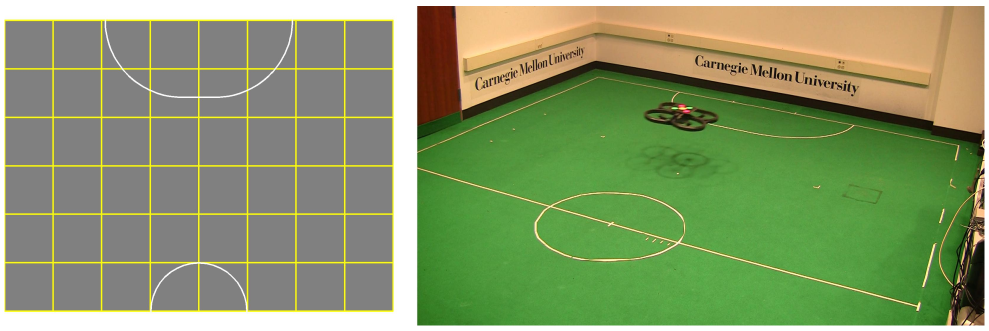
We have described specific versions of some of the components of creating our visualizations above, but there is room for change in both masking and coordinate transformation. Our implementation separates the components so that different versions of each can easily be provided and combined into the pipeline.

We create a mask for a prerecorded video by preprocessing the video to create a new video file where all pixels are white (representing the foreground) or black (representing the background). The viewer and drawing software then read this file alongside the original video to draw with the correct mask. Anyone can use any video-to-mask algorithm to provide input to the rest of the system, without even needing to use the same programming language. A change to coordinate transformation to use more general transformations would be more involved and integrated within the existing code, but the code is structured such that the necessary changes would be contained within one particular part of it.

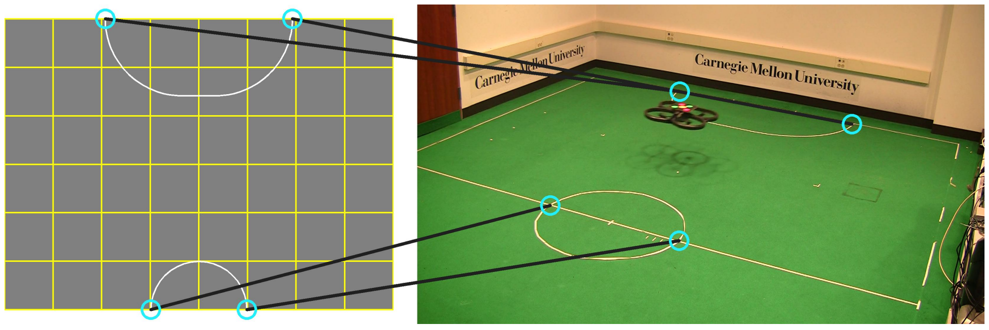
3.4 Quadrotor revisited

Now that we have discussed the general robot-independent form of our visualizations, we can revisit the quadrotor problem and see how to instantiate the general visualization machinery for it.

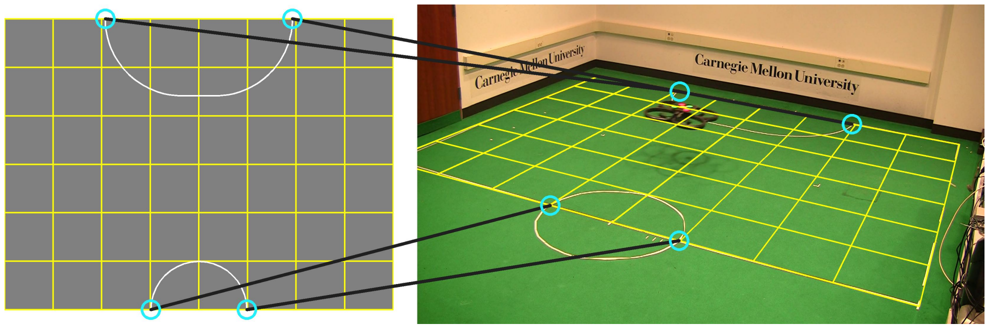
As shown in Table 2.1, we have already written out how the quantities from the quadrotor calculations map onto drawings as we have shown them, so we can directly translate the quantities



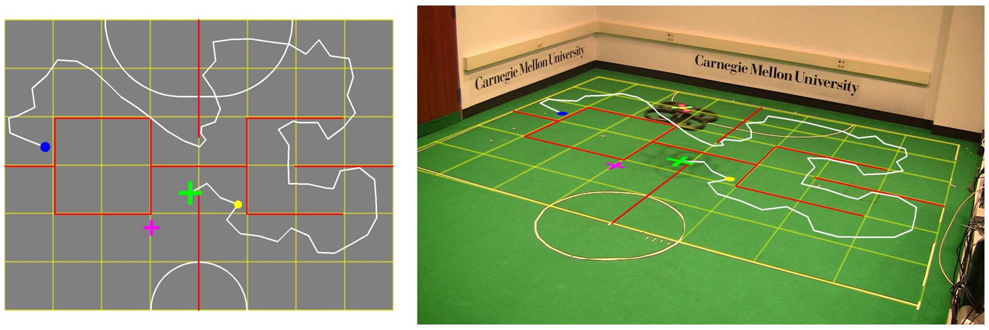
(a) We have a coordinate system, represented by the grid on the left, that corresponds to the rectangle bounded by white lines in the image on the right, and we want to map from those coordinates into the coordinates of the image.



(b) We find at least four point correspondences between the ground coordinates and the image coordinates.



(c) From the individual correspondences, we can compute a homography, which maps *any* point in one coordinate system to the other.



(d) Finally, we take the actual drawings and map their coordinates into the image accordingly.

Figure 3.6: A demonstration of the process of mapping the coordinates from world coordinates in a log to image coordinates in a frame.

Inputs:

- a homography (implicit) that ApplyHomography applies to its argument
- an image array (implicit) that DrawImageLine modifies by drawing
- *color*: the color to use for the drawing
- *p*, *q*: the endpoints of a line segment to draw
- *vertices*: a list containing the vertices a polygon to draw
- *center*, *radius*: the center and radius of a circle to draw

Constants and helper functions:

- ApplyHomography(p): the result of applying a homography to the point in world coordinates p to map it into image coordinates
- DrawImageLine(p, q, *color*): a function that draws a line on the image in the color *color* between image coordinates p and q
- *CirclePoints*: the number of points we sample from the boundary of a circle when drawing

```
1: function DrawLine($p$, $q$, $color$)
2:     $p' \leftarrow$ ApplyHomography($p$)
3:     $q' \leftarrow$ ApplyHomography($q$)
4:     DrawImageLine($p'$, $q'$, $color$)
5: end function

6: function DrawPolygon($vertices$, $color$)
7:     for $i = 0, \ldots,$ Length($vertices$) $- 1$ do
8:         DrawLine($vertices[i-1]$, $vertices[i]$, $color$)
9:     end for
10: end function

11: function DrawCircle($center$, $radius$, $color$)
12:     for $i = 1, \ldots, CirclePoints$ do
13:         $\theta \leftarrow (i-1) \cdot 2\pi / CirclePoints$
14:         $\phi \leftarrow i \cdot 2\pi / CirclePoints$
15:         $p \leftarrow center + radius \cdot \langle \cos(\theta), \sin(\theta) \rangle$
16:         $q \leftarrow center + radius \cdot \langle \cos(\phi), \sin(\phi) \rangle$
17:         DrawLine($p$, $q$, $color$)
18:     end for
19: end function
```

Algorithm 5: The algorithms for drawing a set of shapes onto an image, given the necessary masking and transformation information.

from their initial representations into drawings conforming to the grammar of Figure 3.1.

Recall that instantaneous frames have one timestamp and are good for quantities that change quickly, while persistent frames for information that stays around for longer. The obstacles stay around permanently, in this domain, and the target stays in one place for a while at a time, so those are good choices for persistent frames. Everything else changes every time the quadrotor runs its algorithm, so the rest of the information goes into instantaneous frames.

Figure 3.7 shows how some of the drawings of Figure 2.11 map to entries in a log. The timestamps there are times since the beginning of execution. The portion of the log shown divides into three groups: (a) a persistent frame that describes the static obstacles and is relevant for all time, (b) a persistent frame that is relevant for as long as the quadrotor's target is the same as its current target, and (c) an instantaneous frame containing the results of one iteration of planning. The full log would contain other groups for the different targets, and many more groups for the other planning results.

3.5 Discussion and summary

In this chapter, we laid out our framework and methods for creating visualizations of reasoning for autonomous robots. We generalized the specific example domain described in Chapter 2 to show how we will be able to apply similar concepts to other domains. To do so, we introduced a general structure for storing the drawings generated by autonomous robots, suitable for handling output from any robot, as well as a concrete implementation of the structure and software for viewing it. We provided more detail on the drawing process, and finally demonstrated how the new structures of this chapter apply to the quadrotor domain. In the following chapters, we will describe the other robotic agents that we have contributed and how we can apply the same visualization methods to them.

```
{ { {-inf; inf};
    {
      { { line: (-3025, 500); (-1600, 500)}; red; {obstacle; 0} }
      { { line: (-1600, 500); (-1600, -500)}; red; {obstacle; 0} }
      { { line: (-1600, -500); (-3025, -500)}; red; {obstacle; 0} }
      { { line: (-3025, -500); (-3025, 500)}; red; {obstacle; 0} }
      { { line: (0, 500); (-1400, 500)}; red; {obstacle; 0} }
      { { line: (-1400, 500); (-1400, -500)}; red; {obstacle; 0} }
      { { line: (-1400, -500); (0, -500)}; red; {obstacle; 0} }
      { { line: (0, -500); (0, 500)}; red; {obstacle; 0} }
    } }
  { {15.32; 35.68};
    { { { circle; (700, 1200); 50}; blue; {target; 0} }
    } }
  { 27.56; {
      { { circle; (2271, -1136); 70}; green; {position; 0} }
      { { circle; (1873, -1046); 30}; pink; {acceleration; 0} }
      { { circle; (1412, -568); 40}; yellow; {point; 0} }
      { { line; (2271, -1136); (1412, -568)}; white; {path; 0} }
      { { line; (1412, -568); (1475, -378)}; white; {path; 0} }
      { { line; (1475, -378); (1537, -188)}; white; {path; 0} }
      { { line; (1537, -188); (1590, 5)}; white; {path; 0} }
      { { line; (1590, 5); (1576, 205)}; white; {path; 0} }
      { { line; (1576, 205); (1444, 355)}; white; {path; 0} }
      { { line; (1444, 355); (1525, 477)}; white; {path; 0} }
      { { line; (1525, 477); (1399, 632)}; white; {path; 0} }
      { { line; (1399, 632); (1244, 758)}; white; {path; 0} }
      { { line; (1244, 758); (1089, 884)}; white; {path; 0} }
      { { line; (1089, 884); (964, 1041)}; white; {path; 0} }
      { { line; (964, 1041); (886, 1225)}; white; {path; 0} }
      { { line; (886, 1225); (704, 1309)}; white; {path; 0} }
      { { line; (704, 1309); (700, 1200)}; white; {path; 0} }
    } } }
```

Figure 3.7: The text representation of the frame of the log corresponding to the drawings in Figure 2.11. The first group, with the timestamps of `-inf` and `inf` (representing an interval containing all time) describes the static obstacles in the environment; the second describes the target as long as it is in its current location; the last group describes the robot state at the timestep depicted by the video frame.

Chapter 4

RoboCup SSL

In this chapter, we provide some general background information about the RoboCup robotic soccer competition, and the RoboCup Small Size League (SSL) in particular, since we developed many of the applications and algorithms in this thesis in the context of the SSL. In Section 4.1, we discuss the aims of the RoboCup competition and the various leagues within it; in Section 4.2, Section 4.3, and Section 4.4, we begin to focus on the SSL, describing its physical and software components and the interaction between them. In Section 4.5, Section 4.6, and Section 4.7, we move on to the rules of the game and their enforcement during gameplay. Finally, in Section 4.8, we return to the theme of visualization by providing an example of an algorithm within the SSL that we can fruitfully visualize.

4.1 RoboCup and the SSL

The RoboCup SSL promotes research in multi-agent coordination in real-world adversarial domains. In the SSL, teams of six robots play a scaled-down version of FIFA soccer with a golf ball in a field of size $9\,\mathrm{m} \times 6\,\mathrm{m}$. Each robot is controlled via radio commands sent by its team's offboard computer. Four overhead cameras observe the field and detect the positions of the robots and ball with high frequency and fidelity; as compared to other RoboCup leagues, the global vision and centralized planning mean that teams focus on hardware development, coordination behaviors, and high-level teamwork strategy, as opposed to perception, locomotion, or methods of distributed planning. Games in the SSL are fast-paced, with the robots traveling up to $3\,\mathrm{m/s}$ and kicking the ball at speeds of up to $8\,\mathrm{m/s}$.

The overarching target of the RoboCup Federation, which includes several other soccer leagues, based on widely varying robot types, as well as some non-soccer competitions, is

> By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup[1].

The various leagues are intended to collectively spur development in the different areas of development that will ultimately be necessary to produce such a team. The humanoid soccer leagues, for example, focus on bipedal walking and ball manipulation; the SSL focuses on

[1] http://www.robocup.org/about-robocup/objective/

centralized planning and fast reactions in a dynamic environment; the Middle Size League (MSL), which uses roughly human-sized wheeled robots and a normal soccer ball, focuses on decentralized planning and sensing without dealing with the difficulties of legged robots; finally, the 2D and 3D simulation soccer leagues use simplified models of the world to allow teams to focus on strategy and artificial intelligence without having to maintain hardware.

In the SSL in particular among all the leagues, the fast pace and small ball lead to the recurring problem that not many spectators are able to see the action in any detail. Here and there throughout this chapter, we will begin to see how the visualizations of this thesis can help display the behaviors and actions of an SSL game.

CMDragons: Carnegie Mellon's SSL team Carnegie Mellon University has maintained a team for the SSL since the league's inception, currently called CMDragons (named for Carnegie Mellon University and the logo of its School of Computer Science, which depicts a dragon). For the remainder of this thesis, we will focus on CMDragons as an example of the software and hardware used by the league.

4.2 Physical components and communications

Although there are few restrictions on the physical design of SSL player robots, primarily that each one must fit within a cylinder of diameter 180 mm and height 150 mm, most teams have converged toward similar designs; Figure 4.1 demonstrates the design used by CMDragons[2], and Figure 4.2 shows a typical SSL match at the RoboCup tournament. The robot has an omnidirectional drive system based on four omniwheels. For ball manipulation, it has a dribbler, a horizontal rubber-coated bar which rotates to put backspin on the ball, allowing the robot to hold the ball for some time while driving around. Finally, the robot has both a flat kicker, which kicks the ball straight forward at high speed, and a chip kicker, which kicks the ball at an angle of about 45° above horizontal. Use of the chip kicker enables robots to pass or clear the ball even when there are opponents in front of them.

The ball is a typical golf ball, colored bright orange for ease of detection by the vision system. According to the rules, it should be approximately 43 mm in diameter and have a mass of approximately 46 g.

The overhead cameras are mounted on scaffolding constructed around and above the field. The league has been using FireWire 800 cameras at competitions, though there is a move to Ethernet cameras in progress. All of the cameras are connected to a neutral computer running the SSL Vision System (SSL-Vision) [Zickler et al., 2009], which processes the images from the cameras to extract the positions of the robots and balls on the field. Each camera transmits sixty frames per second to SSL-Vision; each time SSL-Vision receives a frame from any camera, it broadcasts the detected positions from that frame to the team computers. Note that SSL-Vision does *not* transmit (and therefore we as teams do not have access to) the original images from the cameras. However, team members and spectators often set up their own video cameras next to the field to record the game; we work with the videos produced by these external cameras.

[2]We thank Mike Licitra for the mechanical design of the robots and for the design and fabrication of the robot electronics.

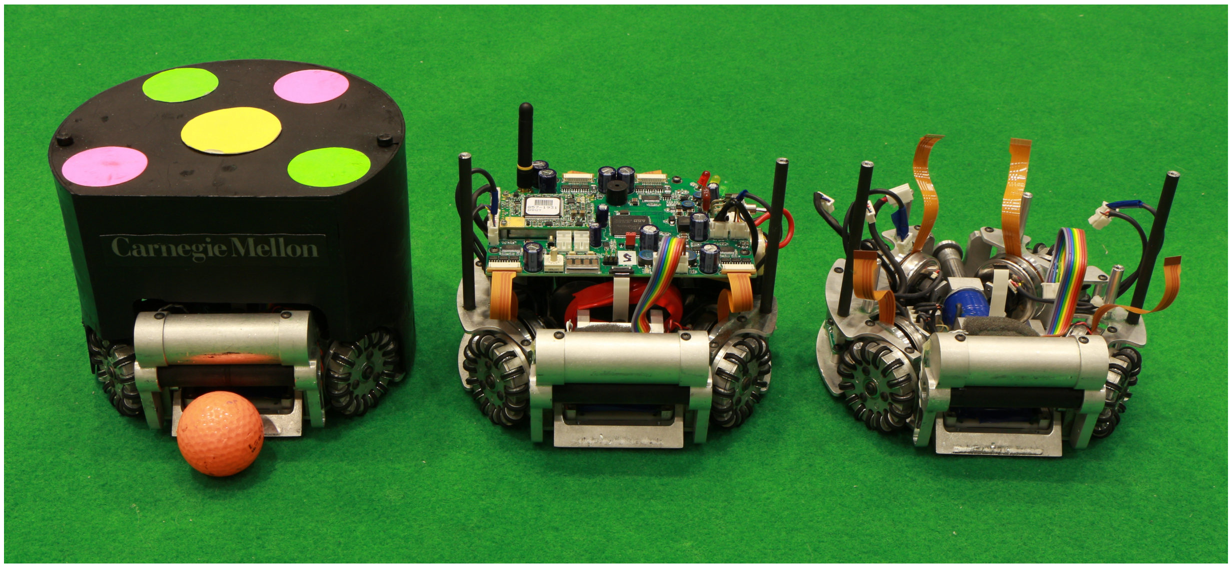


Figure 4.1: The robots of CMDragons, showing (left) a full robot with a ball on its dribbler, (middle) a robot without its top cover, and (right) without its main electronics board.

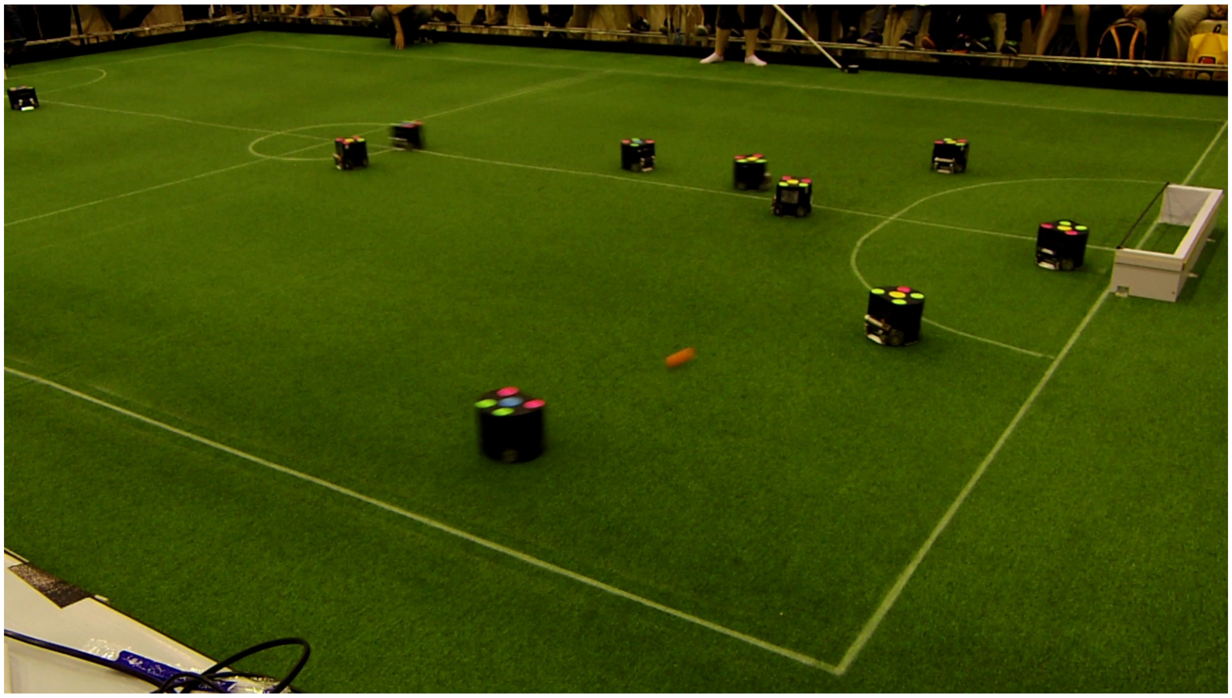


Figure 4.2: A representative moment from the RoboCup 2015 tournament, showing an indirect free kick taken by CMDragons.

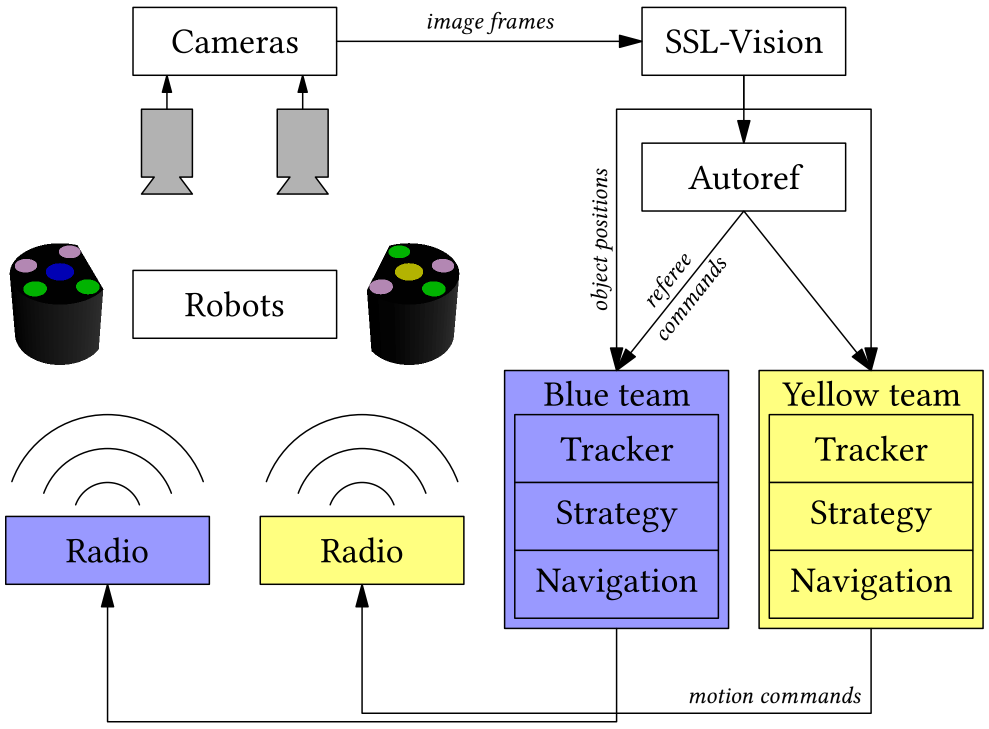


Figure 4.3: A schematic diagram of the components of an SSL game. The two teams, their robots, and the neutral cameras together make up one largely autonomous system.

The referee's decisions are transmitted to the teams via another computer running a specialized program, the *referee box* (or refbox). To ensure low latency and avoid bandwidth constraints between the team computers, referee computer, and vision computer, all of them are connected via a wired Ethernet network specific to the playing field.

Figure 4.3 shows a simplified overview of the overall connections between the components.

4.3 Data acquisition

There is a great deal of technical machinery that is needed to bring the data about the physical robots in a game to the software. Understanding it is helpful for providing context for how the whole SSL ecosystem fits together. In this section, we describe the components that allow game-playing teams in the league to follow along with the state of the world, as they are a critical part of the flow of information through an SSL system.

4.3.1 SSL-Vision

In this section, we will give a brief description of how SSL-Vision computes robot and ball positions based on camera input [Bruce and Veloso, 2003, Bruce et al., 2000, Zickler et al., 2009]. SSL-Vision processes data from four overhead cameras, each operating at 60Hz, and transmits data from every frame to the teams as soon as it is available.

One part of the setup of SSL-Vision is geometry calibration. Since it would be impractical to require that the cameras be precisely placed in particular positions relative to the field, it is necessary to place them approximately first and determine their positions after the fact. For each camera, a human user must first input:

- the height of the camera above the ground;
- the image coordinates of several points on the field with known physical coordinates, which are typically the intersections of field markings; and
- descriptions of the field markings (described as line segments or arcs) within the field of view of the camera.

From the first two of these inputs, SSL-Vision constructs an initial guess at the transformation between field and image coordinates by finding the camera parameters that minimize the sum of squared errors between the given image coordinates and the projections of the world coordinates. Next, it alternates between (a) applying edge detection to the image to extract points which may be on the field markings, assuming that the current camera parameters are correct, and (b) refining the guessed parameters, assuming that the points are actually on the markings.

The edge detection is performed only within a neighborhood of the projected coordinates of the described field markings, and each detected point is associated with the edge in whose neighborhood it was found. The refinement is then done by adding another term to the error function for each detected point, consisting of the minimum squared error between the detected point and the projection of any point on the marking.

The other part of setup is color calibration. For each camera and each relevant color, a human uses a graphical interface to select a set of bins in YUV color space that should be classified as that color when seen by that camera. The relevant colors are orange, for the ball, along with blue, yellow, pink, and green, for the tops of the robots. SSL-Vision also allows classification of a separate green for the field surface and white for the field lines, but those colors are not currently used during the detection process.

After geometry and color calibration, SSL-Vision is ready to begin working. Each frame received from a camera goes through three main stages of processing: color thresholding, blob detection, and pattern detection. In color thresholding, each pixel is classified according to the bins defined in the color calibration step. In blob detection, contiguous regions (“blobs”) of pixels that have been classified as the same color are identified, and their bounding boxes are computed. At this point, any sufficiently large blobs of orange pixels are identified as balls. Finally, in pattern detection, SSL-Vision finds nearby blobs that are arranged in a way that matches the shape of the standard robot pattern, and combines the positions of the blobs to compute an overall position and orientation of the detected pattern. The “butterfly” shape of the pattern was chosen to minimize the final position error after combining the detected blobs.

See Figure 4.4 for a visual demonstration of the stages of processing.

4.3.2 State filtering

Although the inputs to the teams from the physical game field consist solely of time and the positions of game objects, certain derived values are useful; namely, velocities of the objects and estimates of their positions with reduced error, based on the history of observations. Additionally,

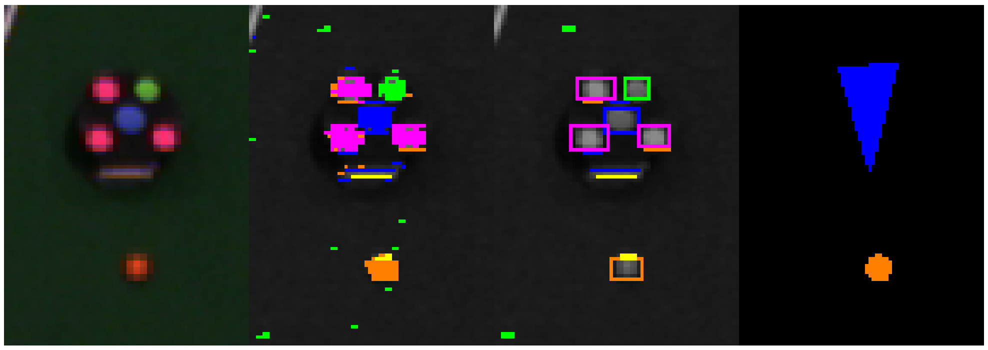


Figure 4.4: A visual demonstration of the stages of SSL-Vision processing. From left to right: (a) a raw image from a camera, (b) the result of per-pixel color thresholding, (c) the result of blob detection, and (d) the result of pattern detection (symbolically representing the computed coordinates of the ball and the coordinates and orientation of the robot).

SSL-Vision reports data from each camera independently; cameras may be out of sync, or multiple cameras may be able to see the same object, and it is preferable to have a fused view where each update includes new information from every camera and multiple views of an object are combined. We briefly discuss here the filtering process used by CMDragons.

To compute the necessary derived information, we apply a discrete-time extended Kalman filter [Simon, 2006, p. 407] to the raw SSL-Vision values. A (plain) Kalman filter takes in a sequence of noisy observations of a system whose state vector evolves over time with known linear dynamics, and with a linear mapping from states to observations (along with additive noise). It combines the observations to produce estimates of the state with lower error than the individual observations. Essentially, it functions by performing Bayesian updates on a Gaussian belief state according to the dynamics and observations.

An *extended* Kalman filter generalizes a Kalman filter to use nonlinear dynamics by linearizing the dynamics around the current state; *discrete-time* refers to the property that the dynamics are a discretization of some underlying continuous-time dynamics.

The dynamics of the filter in our team's implementation take into account factors specific to the SSL, including the commands sent to the robots, when available, and collisions between objects.

4.4 Simulation

The separation enforced by the network means that individual components can be switched out without disrupting the rest. Of course, this ability is used in playing different pairs of teams against one another; it doesn't matter to one team if the other team is replaced by a different one. But SSL-Vision itself can also be replaced by something else. CMDragons, and many other teams, often use a soccer simulator to fill in that place. Our simulator uses Nvidia's PhysX engine[3] to realistically simulate a physical ball and robots reacting to soccer commands [Zickler, 2010].

[3]https://developer.nvidia.com/physx

4.5 Human refereeing

Currently, the rules of the game are enforced much as they are in FIFA soccer: a human referee and assistant referee watch the game, constantly checking whether any rule has come into effect. The referee signals for a stoppage by blowing a whistle, and verbally calls out the subsequent state of the game (e.g. “indirect free kick for yellow”); a human at a referee computer near the field then enters the commands into a specialized referee box program, which transmits them to both teams. Figure 4.5 depicts the refbox interface. The commands and their meanings are defined by the league’s technical committee[4]; some of the most commonly used ones are described in Table 6.1. If `TEAM` appears in the name of an entry there, it actually corresponds to two commands, one for the blue team and one for the yellow team. The refbox and the predefined commands make up the primary interface between humans and robots in the game, and provide a powerful and efficient way to transfer the necessary judgments to the teams.

Apart from announcing the calls, the most important job of the human referees is to position the ball for each kick. The rules specify a point from which each free kick must be taken; the referees use sticks to position the ball appropriately while the game is stopped.

4.6 Game structure

Like in human soccer, the period of normal gameplay consists of two halves; in the SSL, each half consists of ten minutes of play. Game time is broken up into episodes of active gameplay punctuated by times when the game is temporarily stopped.

During an episode, the players may interact with the ball, more or less at will, and they attempt to do so in order to get it into the opponent team’s goal. An episode ends, and play stops, when the ball exits the field or a player commits a foul.

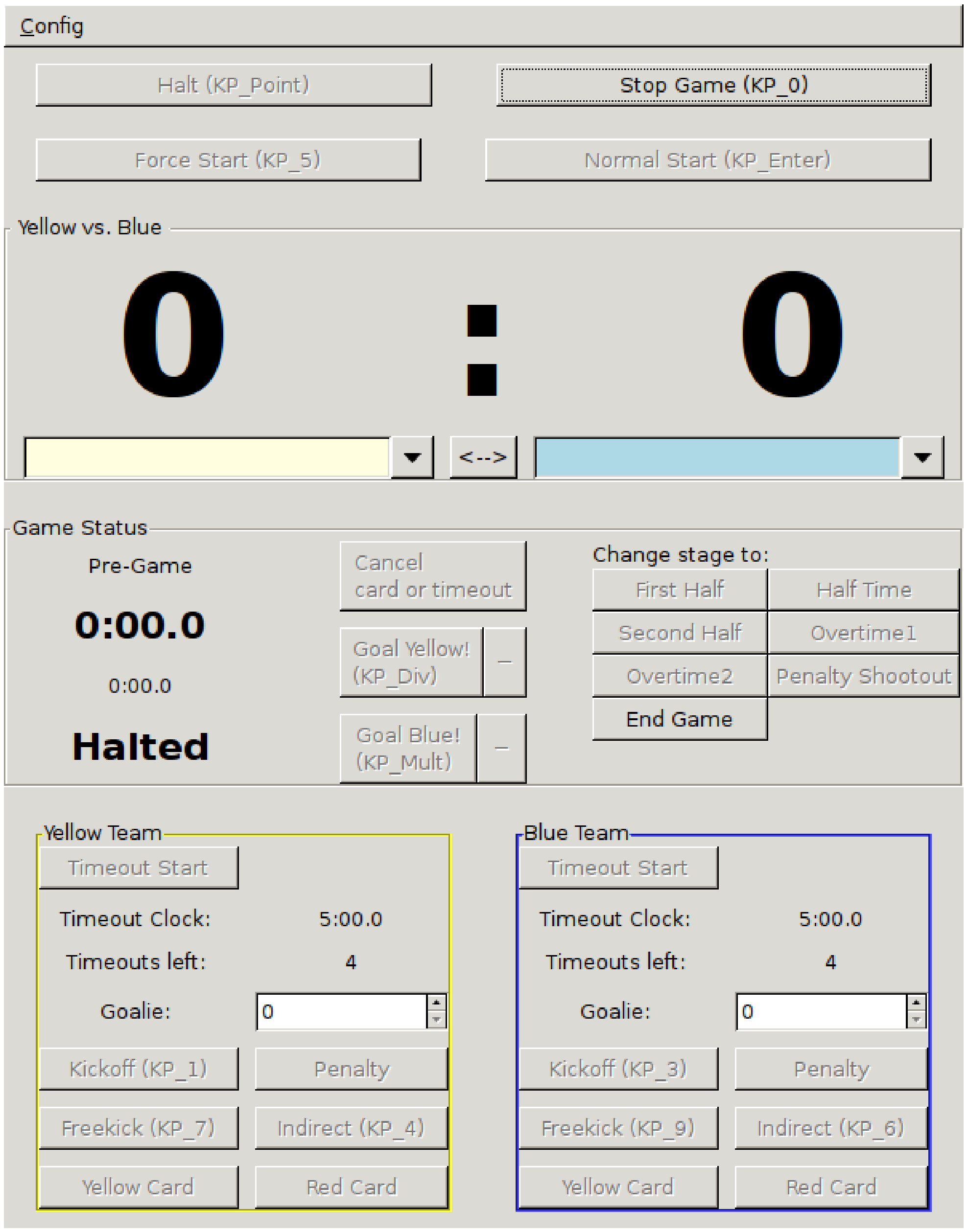
When play stops, the rules specify the type of kick with which to start the next episode and a point from which the kick should be taken. While play is stopped, the robots must move at less than 1.5 m/s and remain at least 500 mm away from the ball, so that the referee may position the ball. Once the ball is in place, the referee will typically wait up to a few seconds for the robots to reach a steady state, and then call out the appropriate signal to the human at the refbox. (Waiting for the robots in this way is not specified in the rules, but is, in our experience, done to some degree by every human referee.)

Most kinds of kicks are to be taken by a specific team; until a member of that team takes the kick, all members of the other team must continue to remain 500 mm away from the ball. Once the kick is taken, the next episode begins.

If the appropriate team has not taken the kick 10 seconds after the command indicating the kick is sent, the referee calls for another stop, followed by a *neutral start*; when the neutral start is called, active gameplay starts immediately.

This structure means that any point in a game naturally falls into one of a few states, and that transitions between these states occur in a predictable fashion: the game might be actively running, or it might be stopped, or it might be waiting for a kick to be taken. In each of these cases, what the robots are doing and what the referee is looking for are qualitatively different

[4] `http://robocupssl.cpe.ku.ac.th/referee:protocol`



Pre-Game

0:00.0

0:00.0

Halted

Yellow Team

Timeout Clock: 5:00.0

Timeouts left: 4

Goalie: 0

Blue Team

Timeout Clock: 5:00.0

Timeouts left: 4

Goalie: 0

Figure 4.5: The interface displayed by the refbox when it starts up. Most of it is taken up by buttons corresponding to all the possible commands, which are automatically activated when it might be appropriate to transmit them; the refbox also keeps track of timing and game state information to act as an aid to the human user.

from what it is in the other cases. This leads into the idea of representing the game as a hybrid automaton, which is discussed in Section 6.2.

4.7 Rules

The rules of the SSL, as described by the technical committee's official rules document [Committee, 2015], fall into a few main categories: time-based, robot-based, and ball-based. In this section, we discuss the most prominent rules of each category.

4.7.1 Time-based rules

The conceptually simplest rules are those relating to game time. Each half of the game lasts ten minutes, with a five-minute half time in between. Time starts counting when the `READY` command is sent to teams at the beginning of a half. Teams may also call for a timeout while play is stopped, which halts the countdown of time and allows team members to handle robots and use their computers; per game, each team may take up to four timeouts with a total duration of up to five minutes.

4.7.2 Robot-based rules

There are a number of rules that are triggered by the interactions between robots themselves. Individual robots are easier to see and keep track of than the ball, which helps make these rules easier to check, but those factors are countered the large number of robots and by an element of subjectivity in some rules, such as the first one listed below.

- If a robot is found guilty of "unsporting behaviour" or "serious and violent contact", (typically, colliding with a slower-moving opponent) an indirect free kick is awarded to the opposing team from the point where it happened.
- If a robot touches the opponent goalie with the point of contact inside the goalie's team's defense area, an indirect free kick is awarded to the opposing team from the nearest legal free kick point.

4.7.3 Ball-based rules

Finally, some rules are based on what happens to the ball, or on interactions between the ball and the robots; there are limitations on how robots may interact with the ball. Although these rules are straightforward to describe, checking them is complicated by the facts that vision of the ball is often lost while it is near a robot (both by human referees and SSL-Vision), just when these rules might need to be invoked, and that chip kicks interfere with knowledge of the true position of the ball (but possibly more so for automated systems; see Section 6.3.3).

- When the ball exits the field without entering a goal, a free kick is awarded to the team which touched the ball less recently. When the exit point is on a touch line (a side of the field not containing a goal), an indirect free kick is awarded from the point 100 mm from the touch line that is nearest to the exit point. When the exit point is on a goal line (a side of the field containing a goal), a direct free kick is awarded; depending on whether the defending or attacking team touched the ball more recently, the kick is taken from the point 100 mm from the nearest touch line and 100 mm or 500 mm, respectively, from the goal line that the ball crossed.

- When the ball enters one of the goals, a goal is awarded to the appropriate team, the ball is placed in the center of the field, and the team that was scored on takes a kickoff.
- If the ball is kicked into the air (above the maximum height of a robot) and enters the goal without touching a teammate or transitioning from bouncing to rolling, an indirect free kick is awarded to the opposing team from the point where the kick was taken.
- If a robot kicks the ball at over 8 m/s, an indirect free kick is awarded to the opposing team from the point where the kick was taken.
- If a robot drives more than 1000 mm while touching the ball the whole time, an indirect free kick is awarded to the opposing team.
- If a robot takes a free kick and is the first robot to touch the ball after the kick, (a "double touch"), an indirect free kick is awarded to the opposing team from where the touch occurred.
- If a robot touches the ball while partially inside its own team's defense area, a yellow card is given to its team; if it touches the ball while entirely inside the defense area, a penalty kick is awarded to the opposing team.

A team may also be shown yellow or red cards for a variety of other offenses, such as if it "is guilty of unsporting behaviour", "is guilty of serious and violent contact", "persistently infringes the Laws of the Game", or several others.

Red and yellow cards both decrease by one the number of robots that the receiving team is allowed to have on the field; a red card remains in effect for the remainder of the game, while each yellow card remains in effect for two minutes of game time. (Multiple yellow cards may be in effect for a team at the same time; their effects are cumulative, and each one expires two minutes after it is issued.)

4.8 SSL visualization: CMDragons offense

With our overview of the game itself completed, we return to CMDragons in this section in order to give a concrete instance of an algorithm to visualize in the context of the SSL. CMDragons uses the STP algorithm [Browning et al., 2005], which we briefly describe here for some context. At the lowest level are *skills*, behaviors that involve one robot performing a task in isolation, e.g., "drive to a location" or "kick a moving ball". *Tactics* describe a complete behavior for one robot to fill some role on the team, such as "receive a pass" or "work with the goalie to defend the goal". Finally, *plays* describe a set of tactics to assign to the robots on the team, such that they work together to play soccer; different plays perform well under different circumstances. We also use *evaluators*, which perform computations relating to the team as a whole rather than any one robot.

We concern ourselves here with the tactics and evaluator for selectively reactive coordination

(SRC) [Mendoza et al., 2016], which provides part of our team's passing capabilities. SRC positions attacking robots to maximize the probability of receiving passes (and, ultimately, scoring goals). We will build up a visualization of SRC incrementally as we describe the algorithm, much like we did in Chapter 2. The algorithm comprises three main steps:

1. The evaluator chooses a set of zones (rectangular regions of the field).
2. The evaluator assigns each of the zones to an attacking robot.
3. Each attacking robot chooses an action given its zone.

We describe the use of visualizations in this section using instances of the grammar described in Chapter 3. The CMDragons algorithm, of which SRC is a part, receives input and sends commands to the robot at 60 Hz; for each run, the algorithm produces an instantaneous frame showing the results of computation during that run. For each step in building the visualization, we include a pseudocode listing showing how to generate the new drawings in the image from the state of the world and the SRC algorithm.

4.8.1 No visualization

We start off with the plain image in Figure 4.6; the moment it shows is relevant to SRC, but it may not be clear how. With so many robots in a small image, it can be hard to tell at a glance even basic information such as what team each robot is on, which is an important part of understanding what is happening in the game.

4.8.2 Position visualization

In the image of Figure 4.7, we add to the frame a circle around each robot that shows which team it one is on, as well as an orange circle around the ball; the pseudocode in the figure shows how to create the drawings from the elements of the world state of the SSL game. Adding the drawings shows that the robot holding the ball is on the blue team and is facing toward a teammate, while the other robots in the area are all opponents. Seeing the teams of the robots makes it easier to predict that the robot holding the ball is planning a pass to its teammate.

4.8.3 Zone visualization

Now we begin to show the specific components of the SRC algorithm. Each zone is a rectangular axis-aligned region of the field; at each timestep, the SRC evaluator chooses a set of zones to work with. Human team members choose some possible sets of zones offline; at each timestep, the algorithm may either choose one or generate a set based on the current state of the world. (We will not discuss the algorithms for generating or choosing sets of zones.) The robots that SRC considers available to work with are those running one particular tactic, which we called *ZonePlayer*, in the current play.

SRC designates all but one of the available robots as *support attackers* and assigns them to particular zones; the remaining robot is the *primary attacker*, which attempts to manipulate the ball instead of operating within one zone. To perform the assignment, the algorithm computes, for each robot and each role, the cost of assigning that robot to that role, then finds the assignment

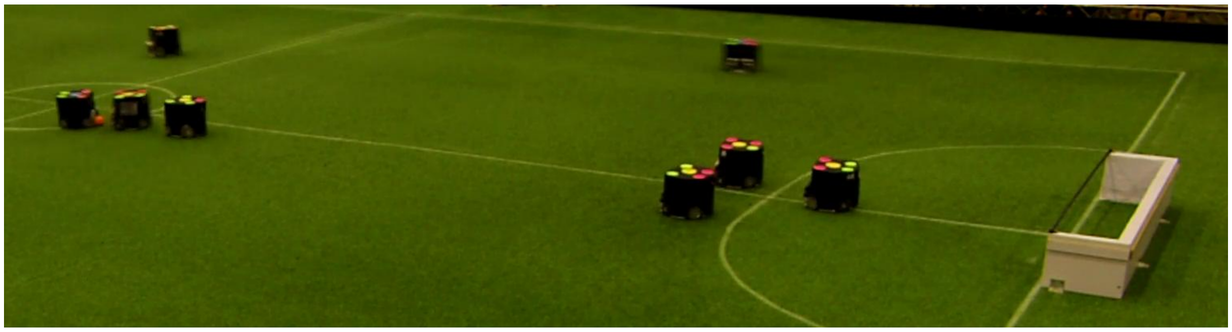
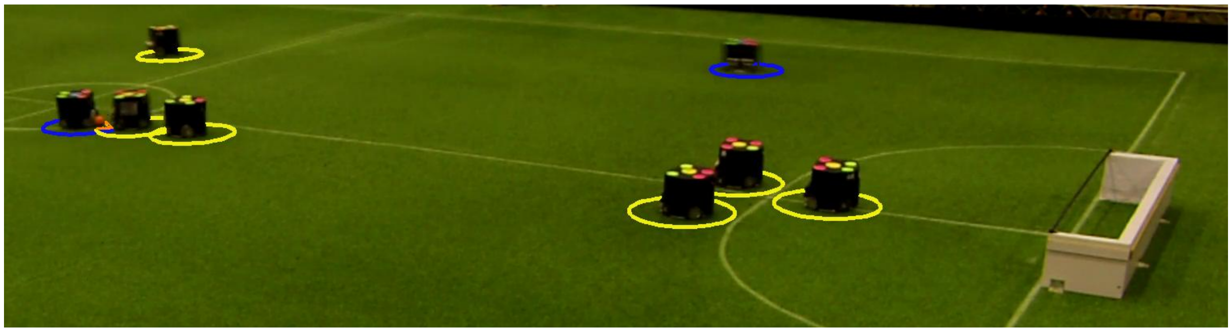


Figure 4.6: A plain video frame showing a time just before a robot makes a pass.



```
Inputs:
  • $t$: the current time
  • $BlueRobots$, $YellowRobots$: the sets of robots on the blue and yellow teams
Constants and helper functions:
  • $ColorBlue$, $ColorYellow$: values representing the colors blue and yellow
  • $TeamBlue$, $TeamYellow$: entity values representing the blue and yellow teams
  • Pos($x$): the position of object $x$
  • $ball$: the ball
 1: function DrawObjects($t$,$BlueRobots$,$YellowRobots$,$ball$)
 2:     for $r \in BlueRobots$ do
 3:         Output($\{t; \{\{\{\text{circle}; \text{Pos}(r); 2 \cdot MaxRobotRadius\}; ColorBlue; TeamBlue; 0\}\}\}$)
 4:     end for
 5:     for $r \in YellowRobots$ do
 6:         Output($\{t; \{\{\{\text{circle}; \text{Pos}(r); 2 \cdot MaxRobotRadius\}; ColorYellow; TeamYellow; 0\}\}\}$)
 7:     end for
 8:     Output($\{t; \{\{\{\text{circle}; \text{Pos}(ball); 50\text{ mm}\}; ColorOrange; Ball; 0\}\}\}$)
 9:     return $frame$
10: end function
```

Figure 4.7: A video frame showing which team each robot is on, along with pseudocode that generates the drawings in the image.

with minimal total cost. The cost for each potential assignment is based on an estimate of how long the robot would take to perform the actions required for the role.

Figure 4.8 shows the zones used by the blue team, which is planning to make a pass, depicted as green rectangles. We use an image showing a larger area than the rest of the images in this section to demonstrate how the zones cover the whole area around the opponent goal with the number of support attackers available.

4.8.4 Action selection visualization

Next in the algorithm, each support attacker computes the point in its zone that would lead to the highest probability of receiving a pass if the pass were to be made to that point. The probability computation depends on factors such as the distance from the point to the goal and the locations of opponent robots that could potentially intercept a pass. Each support attacker then moves to a predefined *default location* within its zone; when the SRC algorithm chooses a support attacker to receive the pass and the primary attacker is ready to pass, that robot actually moves to its chosen location. The receiver delays its motion in order to better conceal the team's intentions, leading to a higher probability of successfully making passes.

Figure 4.9 shows the default locations and specific selected locations for the pass that is about to occur. The green circle shows the planned future location of the center of the robot at the moment the robot receives the pass and the red circle shows the planned location of the ball.

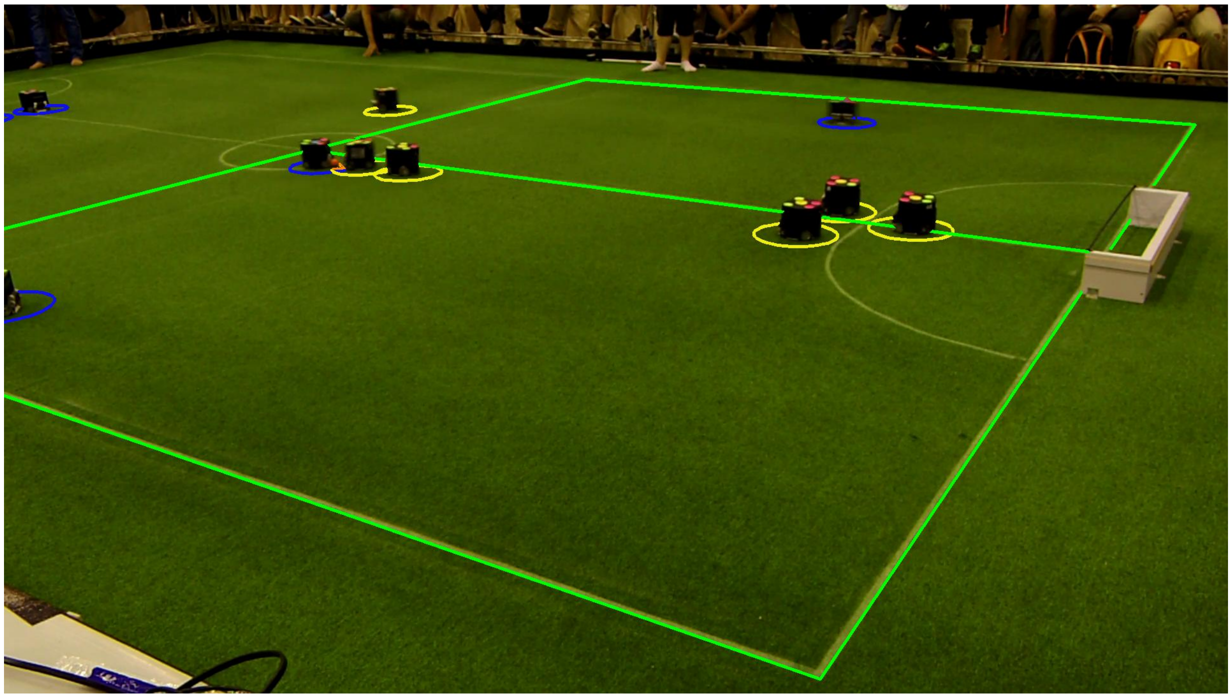
4.8.5 Execution visualization

In order to compute the actual moment to move, the primary attacker (the passer) and chosen support attacker (the receiver) execute a passahead algorithm [Biswas et al., 2014], which chooses a relative timing of the receiver's movement and the passer's movement so that the receiver and the ball arrive at the pass location at nearly the same time, so that the robot can receive the pass while leaving minimal time for the opponents to intercept it.

We omit description of the details of the passahead algorithm, but we can add on some of the quantities that result from the execution, depicting different details of the positions of the receiver robot and ball, as well as the expected trajectory of the ball. We add the following drawings to what is displayed:

- the expected trajectory of the ball from its current location (red line),
- the intended direction for the ball to move after the kick (green line starting from the kicker robot's location), and
- the line from the robot to the ball (blue line starting from the kicker's location).

Figure 4.10 shows the full set of visualizations for the moment when the receiver robot has received the ball and is about to shoot it toward the goal, including the robots, zones, pass locations, and these additional drawings.



Inputs:

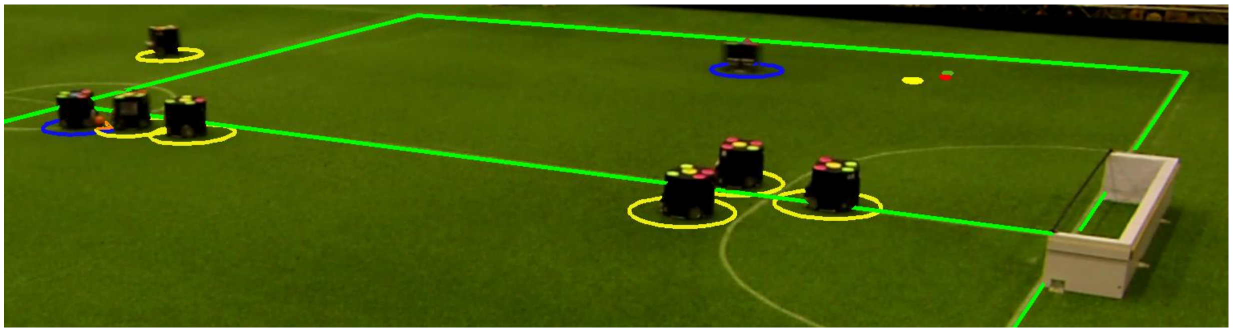
- t: the current time
- *Zones*: the set of zones

Constants and helper functions:

- Pos(z): the center of a zone
- Size(z): the length (x-axis extent) and width (y-axis extent) of the sides of a zone

```
1: function DrawZones($t$, $Zones$)
2:     $frame \leftarrow \varepsilon$
3:     for $z \in Zones$ do
4:         Output($\{t; \{\{\{\text{rectangle}; \text{Pos}(z); \text{Size}(z); 0\}; ColorGreen; Zone; 0\}\}\}$)
5:     end for
6:     return $frame$
7: end function
```

Figure 4.8: A video frame showing the zones in use by the blue team for the pass in progress, demonstrating how the zones cover the area around the opponent goal.



```
Inputs:
  • $t$: the current time
  • $Zones$: the set of zones
Constants and helper functions:
  • $\textsc{DefLoc}(z)$: the default location of the zone $z$

function $\textsc{DrawDefaultLocations}(t, Zones)$
    $frame \leftarrow \varepsilon$
    for $z \in Zones$ do
        $\textsc{Output}(\{t; \{\{\{\text{circle}; \textsc{DefLoc}(z); 90\,\text{mm}\}; ColorYellow; DefLoc; 0\}\}\})$
    end for
    return $frame$
end function
```

Figure 4.9: A video frame showing the zones and default locations in use by the blue team for the planned pass, as well as the specific reception location for the pass.

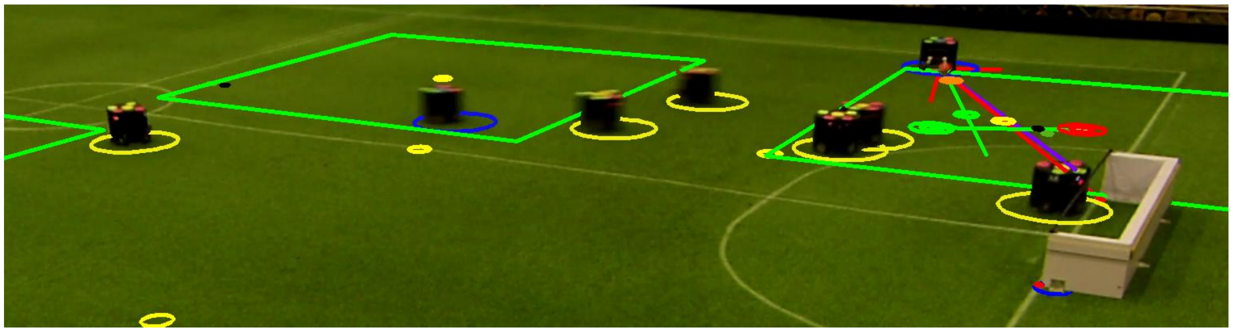


Figure 4.10: A video frame showing a time at the end of the pass, when the receiver has control of the ball and is about to kick it toward the opponent goal. Now that the pass is over and the receiver has the ball, we draw a variety of different shot-related drawings, which we omit detailed descriptions of.

4.9 Discussion and summary

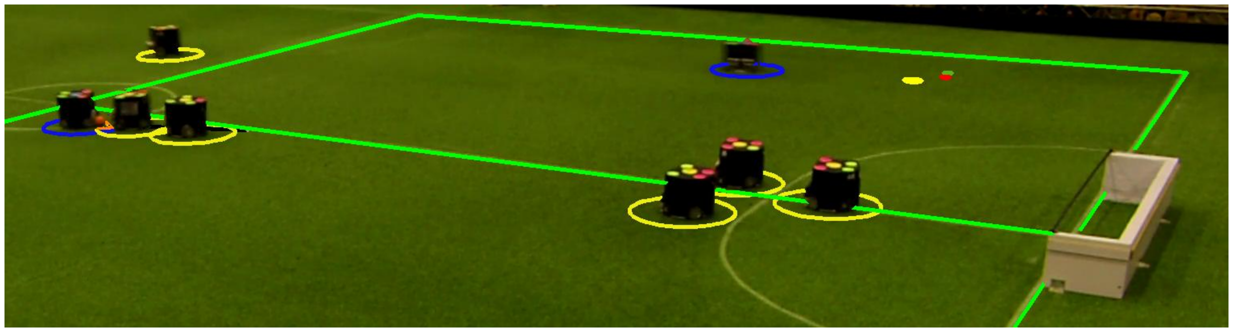
In this chapter, we gave an overview of the mechanics, components, and rules of the RoboCup SSL in order to provide context for the next two chapters, which describe work done within the context of the league. We described both the physical components and the software and conceptual ones, all of which come together to produce the complex behaviors we observe. We also took the opportunity to demonstrate the use of our visualization ideas within the context of the SSL by applying them to an offense algorithm used by CMDragons. In the next chapter, we will do the same with a much more in-depth description of a defense algorithm for CMDragons.

For a planning algorithm such as SRC that runs on real robots, showing aspects of the plan, when they will take time to execute, along with of the current state of the robots blends the future and present aspects of the planning and execution in a way that is impossible to achieve with a plain video alone. Figure 4.11 shows two particularly visible examples of this correspondence across time as demonstrated by SRC, including side-by-side copies of the full versions of the running example of this section. The first image of each pair shows the primary attacker planning to pass to a location on the other side of the image that is marked with the second image shows a point in time a few seconds later, after the intended receiver has received the ball at the planned location.

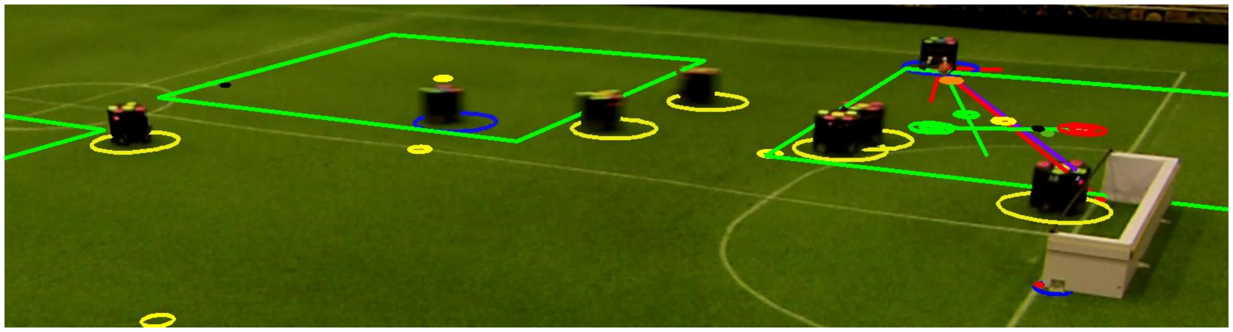
We showed the most important details about the execution of the SRC algorithm. We could also potentially get more detailed, e.g., by showing something about the actual calculations that go into computing support attacker locations, listed here with examples of how they might be shown:

- the location of the robot chosen as primary attacker (a distinctly-colored circle) and
- the assignments of robots to zones (lines from the robots to the default locations within the zones).

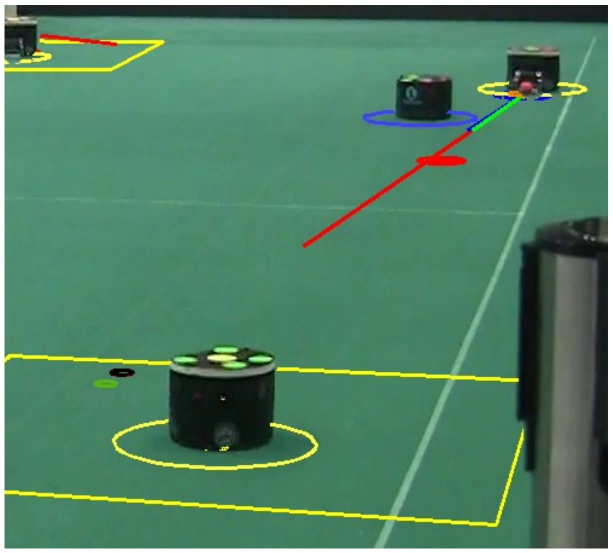
However, the visualization of the algorithm as it was naturally developed for the purposes of the SSL competition did not include that information; as we aimed in this chapter to show the results of a real-world application of visualization to a robot algorithm, we did not include that other information.



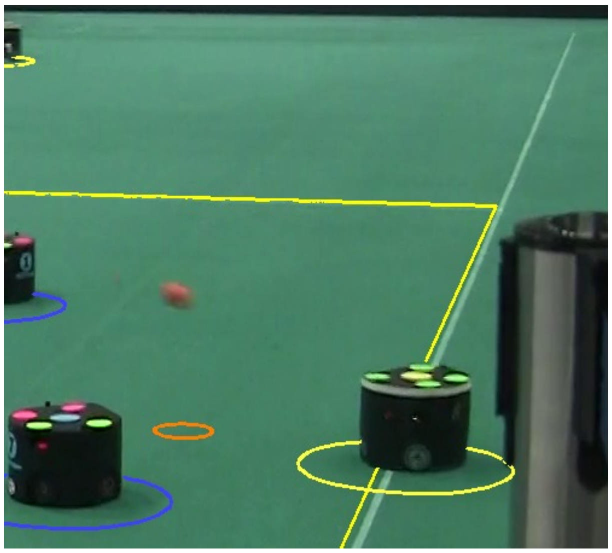
(a) The blue robot on the left plans to pass to its moving teammate on the right, which will receive the ball at the location marked with red and green dots in the top right of the image.



(b) The teammate has received the ball at the planned location and is about to shoot it toward the goal (and score, as it turned out in the actual game).



(c) The yellow robot at the top plans to pass to its teammate lower in the image.



(d) The teammate has received the ball at near the planned location (the difference is due to the inaccuracy of chip kicking) and has kicked it on in turn.

Figure 4.11: Visualizations showing the successful planning and execution of passes by our SSL team.

Chapter 5

RoboCup SSL: threat-based defense

As part of our preparations for the RoboCup tournament, we developed a *threat-based evaluator* to drive the CMDragons defense [Biswas et al., 2014], which we describe in this chapter. Preliminary versions of the defense were developed by Juan Pablo Mendoza and Alex Etling, before the author of this thesis took over development. The defense system works within the context of the STP framework introduced in Chapter 4; it consists of an evaluator that considers how opponents might possibly score on our team's goal in the near future and three tactics that execute actions based on the output of the evaluator.

5.1 Overview

The evaluator considers the positions of the ball and the opponent robots to compute the single *first-level threat* and multiple *second-level threats*, which represent information about ways in which the opponent team might score a goal. The first-level threat represents the most immediate means for the opponents to score a goal, and is either the ball or an opponent robot. When an opponent is about to receive the ball, the first-level threat is that opponent; otherwise, it is the ball. The second-level threats represent possible indirect attacks on our goal; they are the opponent robots besides the one that is closest to manipulating the ball.

The available defenders are then positioned based on the locations of the threats. *Primary defenders*, of which there are usually one or two, move around the edge of the defense area, acting as the line of defense before the goalie; *secondary defenders* move further out on the field, intercepting passes and shots by the opponent earlier on. The primary defenders typically defend against the first-level threat, staying between the ball and the goal; if there are two, but only one is needed to do so, the other will move elsewhere on the defense area to guard a second-level threat. The secondary defenders guard against the second-level threats; each one positions itself on a line either from a second-level threat to the goal (to block an indirect shot) or from the first-level threat to a second-level threat (to block a pass). Except during corner kicks, there are never more than two primary defenders (see Section 5.8), and there are never any secondary defenders unless there are two primary defenders.

We describe the computation in terms of *tasks*: each threat generates one or two tasks, each consisting of one or two positions, a priority, and other auxiliary information; the tasks are then assigned in decreasing order of priority to the available defenders.

First-level threat tasks are always assigned to primary defenders and include one position near the defense area. Second-level threats to block shots include two positions: one near the defense circle, in case the task is assigned to a primary defender, and one further out, for a secondary defender.

Ultimately, the high-level steps of the algorithm underlying the defense evaluator are as follows:

- choose high-level state,
- compute tasks for first-level threats,
- compute tasks for second-level threats,
- assign tasks to primary defenders, and
- assign tasks to secondary defenders.

We expand on these steps in the following sections. The steps describe the core aspects of the defense, which handle the static positioning of the robots to block the avenues of attack by opponents; if the game were static or nearly so, this might suffice to guard our goal. Of course, it is not, so we must take additional measures to account for the dynamic nature of the game; these measures are also described in the later sections of this chapter.

5.2 High-level states

There are several high-level states that the defense engine can occupy, each of which results in qualitatively different behavior by the defending robots.

- The ball is coming toward the defense area and a defender can intercept it using its kicker. That defender will attempt to intercept the ball, while others move nearby and the goalie positions in the projected path of the ball.
- The ball is coming toward the defense area too quickly to be intercepted by a defender. If there are two primary defenders and the ball is coming between them, they will drive toward each other as fast as possible to attempt to block the ball with their bodies.
- The ball is near the defense area and the goalie should clear it, since it is too close for the defenders to manipulate safely. In this case, the primary defenders position themselves to block opponents from reaching it, giving the goalie time to move forward and kick the ball away. See Figure 5.7.
- The ball is inside the defense area. In this case, the primary defenders move around the defense area while the goalie attempts to intercept the ball. If the goalie brings the ball to a stop, it is capable of setting up a pass to attackers on the team.
- The ball is far from the defense area. In this case, tasks are computed normally and the defenders simply move to the positions of their assigned tasks.

5.3 Spatial algorithms

Before we get into the fully defense-specific aspects of the algorithm, we discuss three general spatial algorithms that apply to aspects of the soccer state.

5.3.1 Stadium intersection

Because the primary defenders' tasks are intimately tied to the shape of the defense area, we need geometric algorithms to interact with the defense area. The area has the shape of (half of) a "stadium"; that is, a rectangle with semicircles attached to two opposite sides. Most frequently, we want to be able to find the first point on a given ray that is a given distance from the defense area. The boundary of a stadium can be described as the set of points that are a fixed distance from a particular line segment; therefore, the set of points that are a fixed distance from a stadium is itself the boundary of a stadium. Thus, we simply need to find the first intersection of a ray with the boundary of a stadium.

Initially, we assume that the ray starts outside the stadium. The boundary of a stadium consists of four components: two semicircles and two line segments. We extend the semicircles into circles and the segments into lines, and then analytically find the intersections of the ray with the extensions. Any intersections which are within the extensions rather than the original boundary are discarded. Of all remaining intersection points, the one closest to the ray's start point is returned. Figure 5.1 demonstrates the intersection process for some example situations.

We can also extend the algorithm to allow for starting points inside the defense area. For a starting location $\vec{p}$ and direction $\vec{v}$ defining the ray, we replace $\vec{p}$ with $\vec{p} + D \cdot \vec{v}$, where D is a large scalar, and replace $\vec{v}$ with $-\vec{v}$. For sufficiently large D, doing so effectively finds the first intersection with the stadium when tracing backward along the ray from a point known to be outside the stadium. Since stadia are convex, the original ray must have had only one intersection, so the solution to the altered problem is the original intersection.

5.3.2 Stopping location

Another computation that is used in several places, both in our defense and otherwise, is finding the *stopping location* of a robot (for the defense, always an opponent robot), which is the location at which it would come to rest if were to immediately start decelerating as much as possible. The defense usually uses opponents' stopping location in its computations, rather than their current locations. We assume that opponent robots will generally be able to manipulate the ball only when stopped, and we can therefore look ahead in the future to the point where they might first possibly be stopped.

The stopping point computation uses an estimate of the maximum deceleration of the opponent robots. (We assume, as with our own robots, that the maximum deceleration is independent of direction.) In one dimension, if an object traveling at speed v undergoes a constant deceleration of a, it will come to a stop after a time $\frac{v}{a}$, during which time it has average speed $\frac{v}{2}$, and it therefore travels distance $\frac{v^2}{2a}$ before stopping. On the two-dimensional field, this generalizes naturally, so if a robot has velocity $\vec{v}$ and decelerates (i.e., accelerates in the opposite direction) with magnitude a, its displacement before stopping is $\frac{|\vec{v}|^2}{2a}\widehat{\vec{v}}$.

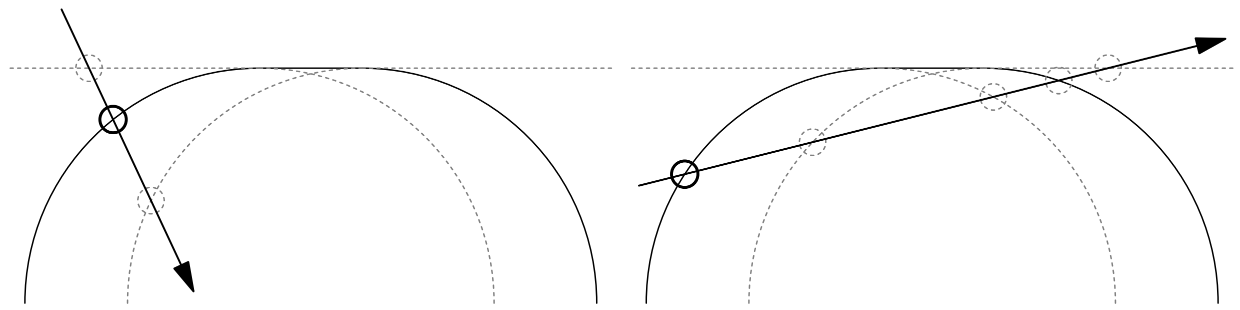


Figure 5.1: A demonstration of the algorithm for intersecting a ray with the boundary of a stadium. (Only half of a stadium is shown, since that is the actual shape of the defense area.) The solid lines represent the boundary and the dashed lines represent the extensions of its components. The solid circle represents the true intersection; the light dashed circles represent intersections of the ray with the components or their extensions that are discarded, either because they lie on the extensions rather than the true boundary or because they are after another intersection.

5.3.3 Velocity estimation

As described in Section 4.3.2, CMDragons uses a Kalman filter to predict the position and velocity of the ball at a time past the latest available observation, in order to account for latency in communication and processing. At RoboCup 2015, however, we noticed that the filter often produced inappropriate results when a robot kicked the ball: instead of quickly updating the velocity to reflect the new trajectory, the velocity changed slowly over the course of about 10 frames (~160 ms). The defense uses the filtered velocity of the ball and to estimate when the ball is approaching our goal, and that time difference can mean the difference between saving a goal and letting it in.

In order to work around this problem, we added the ability for the defense evaluator to compute its own estimate of the ball's position velocity using an algorithm that is less flexible in general, but also less prone to such errors. Algorithm 6 explains that algorithm. The algorithm searches for a good linear fit over ball positions; if it finds one that indicates that the ball is moving quickly toward our goal, it uses those values instead of the ones from the Kalman filter. When it computes an alternate value for the ball state, all defense-related calculations use that value instead of the original values.

5.4 Tasks

A *task* represents an avenue of attack for one defender to guard against. Each task t has the following properties:

- the location on the field where a primary defender would have to position itself to guard against this threat
- the location on the field where a secondary defender would have to position itself to guard against this threat
- whether the task is for blocking a shot on the goal or a pass between opponents

  whether the x-coordinate of the threat location associated with this task is greater than a certain configurable value

Inputs:

- *orig-pos*, *orig-vel*: the tracker's estimation of the ball's position and velocity
- $b_0, b_1, \ldots$: the ball observations, each containing a position and a time, with higher indices indicating earlier observations

Outputs:

- *pos*, *vel*: estimates of the ball position and velocity, possibly different from the original values

Constants and helper functions:

- $\textsc{LinearFit}(b_0, b_1, \ldots)$: the result of performing linear least-squares fit on the x- and y-coordinates of the given observations as functions of time, consisting of a velocity vector, position vector at the current time, and the sum of residuals for the fit
- $\mathit{FitThresh} = 100\,\text{mm}^2$: a threshold for determining whether the linear fit for a set of observations is good enough
- $\mathit{VelThresh} = 2500\,\text{mm/s}$: a threshold on ball velocity for choosing whether to use the fit values
- $\textsc{OutLocation}(p, v)$: the location at which a ray starting at p (assumed to be inside the field) and pointing in direction v intersects the field boundary

```
function OverrideBallVelocity($orig\text{-}pos, orig\text{-}vel, b_0, b_1, \ldots$)
    for $i = 1, \ldots, 10$ do
        $vel', pos', fit\text{-}error' \leftarrow$ LinearFit($b_0, \ldots, b_i$)
        if $fit\text{-}error' < FitThresh$ then
            $vel, pos, fit\text{-}error \leftarrow vel', fit\text{-}error'$
        end if
    end for
    if $|vel| > VelThresh$ then
        $out\text{-}loc \leftarrow$ OutLocation($pos, vel$)
        if Distance($out\text{-}loc, OurGoalCenter$) $< 1.1 \cdot GoalWidthH$ then
            return $pos, vel$
        end if
    end if
    return $orig\text{-}pos, orig\text{-}vel$
end function
```

Algorithm 6: The algorithm that the defense evaluator uses to compute an updated estimate of the ball's position and velocity when the tracker returns unreliable results.

- the robot on our team assigned to this task
- the opponent robot associated with this task
- the point toward which the assigned robot should face when carrying out this task

The defense algorithm can create tasks in a variety of different ways, which we describe in the following sections.

5.5 Primary defenders and first-level threat

The first-level threat is closely associated with the ball position and represents the possibility for the ball to go directly into our goal with a single kick; we defend against it by placing our robots between the location of the threat and the goal. The threat's location is typically the location of the ball itself, but when it appears that an opponent will receive the ball soon, we *snap* the threat to a location on the future trajectory of the ball, which anticipates the point at which the ball will be received, and thus allows the defending robots to position themselves sooner.

Given the threat location, we must then compute where to position robots to guard against it. If possible, we would like to block all open angles to the goal. How the task positions are computed changes depending on whether it is possible to do so, how many robots are needed to do so, and how many robots are primary defenders.

Since primary defenders are the last line of defense before the goalie, they stay close to the edge of the defense area, moving sideways around it to track the ball. Nearly all teams use some equivalent of our primary defenders, robots that defend the goal by moving continuously around the edge of the defense area. This is a sensible tactic due to two aspects of the rules: (a) the rounded shape of the defense area, which enables robots to move smoothly around it, and (b) the rule penalizing a robot for touching the ball after entering its own team's defense area. A perennial topic of discussion in the league is that of changing either or both of those aspects to be more similar to human soccer: changing the area to a rectangle or reducing the restrictions on entering the area — and at the time of writing, the technical committee (TC) had just announced plans to change to a rectangular defense area. Going forward, teams will need new tactics for their equivalents of primary defenders, but the behaviors we describe have worked quite soundly until now.

5.5.1 Threat location

The first step in determining the location of the first-level threat is to estimate which opponent will be able to receive and manipulate the ball first. That robot is not considered to be a second-level threat, since we assume that that opponent is most likely to manipulate the ball directly next, instead of receiving a pass. The first-level threat will either be the ball's location or, if we choose to snap the ball, a point on the ball's future trajectory.

Algorithm 7 shows our algorithm for choosing this robot. The first part of the algorithm keeps track of the nearest robot to the ball, with hysteresis to prevent rapid switching. If the distance from the ball to that robot is small enough or the ball is slow enough, then we choose that robot, as we consider it already in possession of the ball, in the first case, or the best able to move to take possession, in the second case. Otherwise, the ball is in motion and not too close to any robot, so we attempt to estimate which robot can receive the ball soonest as it moves. For each robot, we compute the following value, which we refer to as the *cost of receiving*, as a heuristic estimate for that opponent's ability to receive the ball soon:

$$(1 + S \cdot (1 - c)) \frac{|\vec{p}|}{|\vec{v}_b|},$$

where S is a dimensionless constant, $\vec{v}_b$ is the velocity of the ball, $\vec{p}$ is the vector from the ball's location to the robot's stopping location, and c is the cosine of the angle between $\vec{p}$ and $\vec{v}_b$. This value has units of time and is roughly interpreted as the time for the ball to reach the robot, with a penalty when the robot is not located along the future trajectory of the ball. The value of S, which is 15 in our code, controls the degree of the penalty; Figure 5.2 demonstrates the effect of changing it and the dependence of the cost on robot position for a given ball position and velocity.

Having chosen the closest robot, we then decide whether to snap the threat location to it or leave it at the ball's location; we discuss how to make that decision in the next section. If we do snap the threat, its location is the point on the ray starting at the ball's location and in the ball's velocity that is closest to the robot's stopping location.

5.5.2 Snapping

The main factor in determining whether to snap at each timestep is the relative position of each opponent with respect to the ball.

When this value for the estimated receiver is below a threshold, the first-level threat is, rather than the location of the ball, the closest point on the ball's trajectory to the stopping location of the estimated receiver.

We also tested a method of deciding when to snap based on estimates of the time that it would take each opponent to intercept the ball, computed using our more complex calculations of navigation time for each robot, taking into account its velocity and motion capabilities. However, these estimates can behave erratically, especially when the robot in question is near the ball, leading to unpredictable behavior. Instead, we chose to use the simpler heuristic described above, which captures intuition we developed from observing games and, in our experience, works well.

5.5.3 Defense area margin

All of the tasks for primary defenders in one timestep are positioned at the same distance from the defense area. That distance (which we refer to as a *margin*) is a bounded linear function of the distance d from the first-level threat to the defense area; i.e., the margin is

$$\max(M_0, \min(M_1, A \cdot d + B)),$$

where M_0, M_1, A, and B are constants chosen by hand. For the RoboCup tournament, we used values such that the margin changed from $1.25 \mathit{MaxRobotRadius}$ to $6 \mathit{MaxRobotRadius}$ as the

Inputs:

- *ball*: the ball

Outputs:

- the opponent that we estimate can manipulate the ball soonest

Constants and helper functions:

- $PossessDistance = MaxRobotRadius + BallRadius + 40\,\text{mm}$: a distance threshold for us to consider the robot to be in possession of the ball already
- $SlowSpeed = 250\,\text{mm/s}$: a speed below which the ball is considered essentially stopped
- $HysteresisDistance = 45\,\text{mm}$: a threshold for determining when a different opponent may be considered to become the closest

```
1:  $last\text{-}opponent \leftarrow Null$
2:  function FindFirstLevelThreat
3:      if $last\text{-}opponent = Null$ then
4:          $distance\text{-}threshold \leftarrow \infty$
5:      else
6:          $distance\text{-}threshold \leftarrow \text{Distance}(last\text{-}opponent, ball) - HysteresisDistance$
7:      end if
8:      $opponent \leftarrow \arg\min_{o \in opponents} \text{Distance}(o, ball)$
9:      if $\text{Distance}(opponent, ball) > distance\text{-}threshold$ then
10:         $opponent \leftarrow last\text{-}opponent$
11:     else
12:         $last\text{-}opponent \leftarrow opponent$
13:     end if
14:
15:     if $\text{Distance}(opponent, ball) < PossessDistance$ or $\text{Vel}(ball) < SlowSpeed$ then
16:         return $opponent$
17:     end if
18:     return opponent with lowest cost of receiving
19: end function
```

Algorithm 7: The algorithm for estimating the opponent receiver.

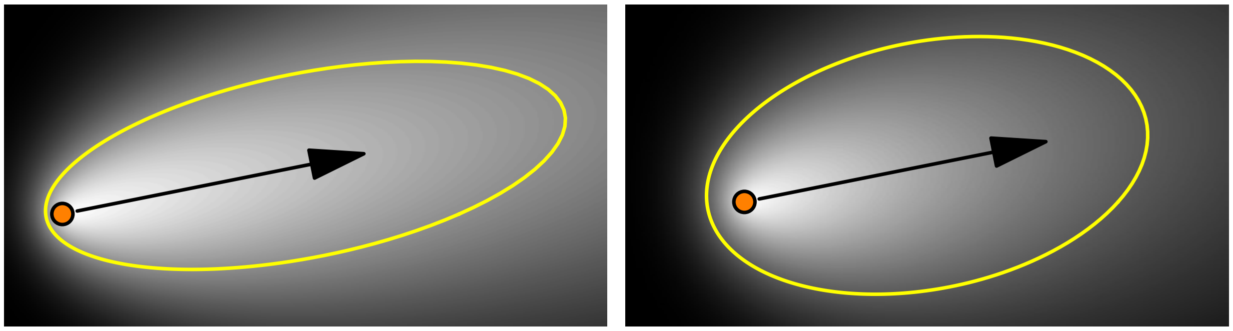


Figure 5.2: The dependence of the cost of receiving on ball position; a lighter color indicates a lower cost. The circle indicates the ball's position and the arrow its velocity. The curve (an ellipse, for our cost function) is a contour along which the cost of receiving is constant. Left: $S = 15$. Right: $S = 5$.

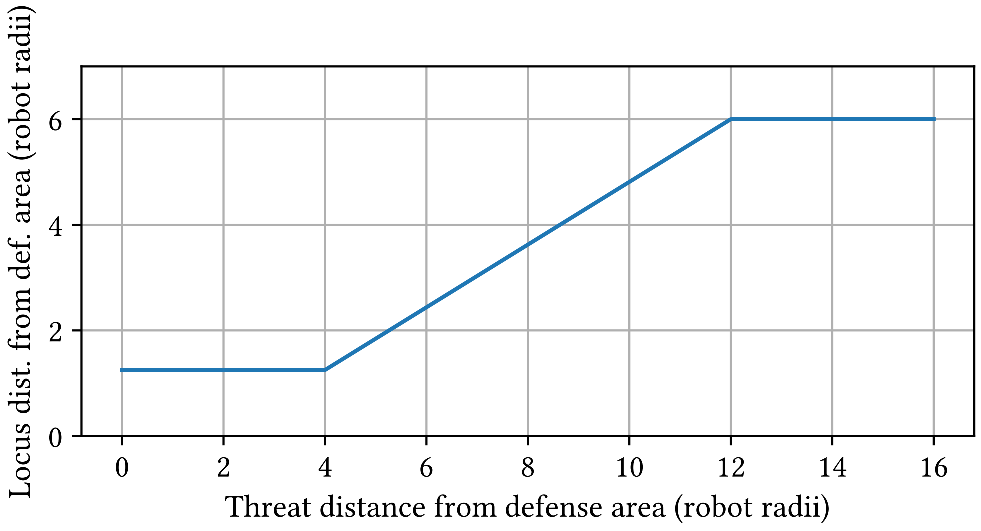


Figure 5.3: A plot of the primary defender margin distance from the defense area as a function of the distance from the first-level threat to the defense area.

threat distance moved from 4*MaxRobotRadius* to 12*MaxRobotRadius*, based on empirical game experience; Figure 5.3 shows the resulting function.

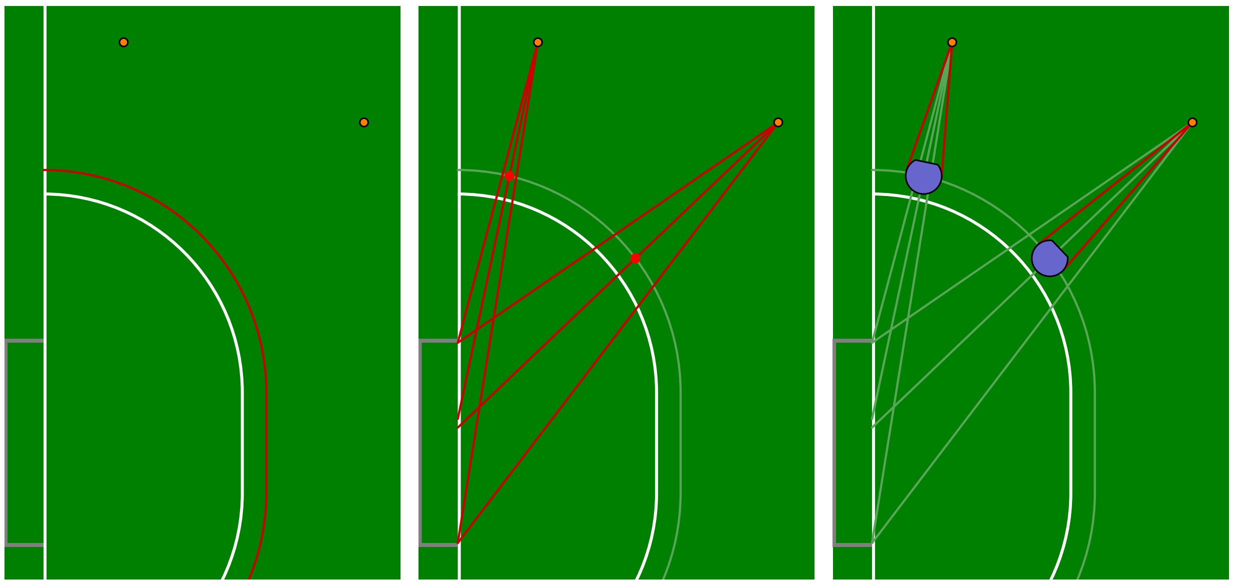
Using a variable margin in this way allows the defenders to move farther forward when the threat is far away, making it faster for robots to switch between offense and defense roles and allowing the defenders to block more of the open angle, while still falling back to stay behind the threat when it gets close to the goal.

For any given margin, the set of points that are within the field and that distance from the defense area is the boundary of a stadium; we refer to this as the *locus* for primary defender positions.

5.5.4 Task positions for primary defenders

When there is only one primary defender, the task positioning depends on whether the open angle to the goal can be blocked by a single robot placed at the current defense area margin. To determine whether this is the case, we compute the bisector of the angle from the threat location to the corners of the goal, and then the intersection of the bisector with the defender locus. If the angle subtended by a robot at that intersection from the threat location is greater than the angle subtended by the goal, then a robot positioned there would block the whole open angle, and that position is chosen to be the only task arising from the first-level threat. Figure 5.4 depicts the steps of this calculation.

When the position so computed cannot cover the whole open angle, we choose a position such that the outline of the robot is tangent to one edge of the open angle. We choose the side of the open angle corresponding to the corner of the goal nearer the threat, based on the idea that doing so leads to a position closer to the threat, so the robot will cover more of the angle. We then rely on the goalie to cover the remaining angle. Figure 5.5 shows how we compute this


(a) Two potential ball locations and the defender locus.

(b) The angles subtended by the goal from each position and the intersections of the bisectors of the angles with the locus.

(c) The angles subtended by robots placed at the intersections.

Figure 5.4: The means of checking whether one defender at the appropriate margin can block the open angle by itself. We demonstrate the steps of the computation for one ball position that a single defender can guard and one that it cannot.

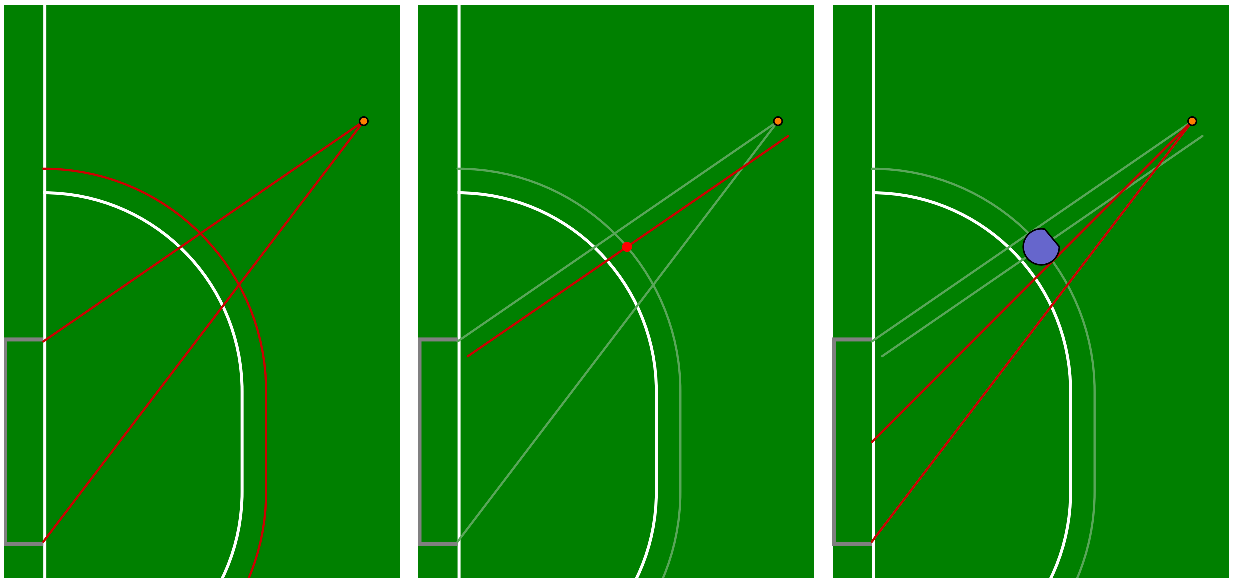
location.

5.6 Second-level threats

So far, we have discussed how to protect against the biggest danger, that of an opponent shooting immediately on the goal, but the other opponents still need to be considered. Even if the direct shot is blocked, the opponents may score by executing a pass followed by a shot, so we must judge which opponents are the most dangerous in this sense. Every opponent robot besides the estimated receiver can potentially be the receiver of such a pass; we call those opponents the second-level threats.

We consider two ways of dealing with each second-level threat: either we may block a pass to it, or we may block a shot from it. Each method generates a task.

Once the positions for the tasks have been computed, we must choose which defender will block each one. To do so, we rank the tasks using a function that compares two tasks according to several criteria that influence how quickly and accurately we expect the opponent robot to be able to shoot on our goal from its position. The tasks that we assign to the defenders are the highest-ranked ones according to that comparison.


(a) The open angle from the position in Figure 5.4 that cannot be covered by one robot, and the defender locus.

(b) The offset line whose intersection with the locus defines the robot position.

(c) The smaller remaining open angle after a robot is placed at the computed location.

Figure 5.5: The means of positioning a single primary defender within the open angle when it cannot cover the whole angle.

5.6.1 Task positions for blocking shots

For blocking shots, we compute a point on the line segment between the threat location and the center of our goal. The calculation of the specific point to choose, an initial version of which was created by Alex Etling, takes into account the estimated speed of the opponent robot and our robot and the total latency of our soccer system to find a point that is far forward as possible while ensuring that a robot placed at that position will not allow an opponent that moves suddenly to have an open angle to the goal. Algorithm 8 shows the computation here; it is straightforward except for the reasoning behind the value of *shift-factor*, which we explain here. We position the defender on the line between *stopping-location* and *OurGoal*, a portion *shift-factor* from the former to the latter.

Suppose that the opponent is stopped at *stopping-location* and facing our goal, then moves sideways at full speed with the ball for time *Latency* and immediately shoots the ball at speed *KickSpeed* toward the goal. The ball would take time *shot-time* (as defined in the algorithm) to reach our goal. To perfectly block the shot, the defender robot would need to move $1-$*shift-factor* as far as the opponent moved in time *shift-factor* $\cdot$ *shot-time*. Assuming it can immediately move at *OurSpeed*, we have

$$\begin{aligned}
(1 - \textit{shift-factor}) \cdot \textit{TheirSpeed} \cdot \textit{Latency} &= \textit{shift-factor} \cdot \textit{shot-time} \cdot \textit{OurSpeed} \\
\textit{TheirSpeed} \cdot \textit{Latency} &= \textit{shift-factor} \cdot (\textit{TheirSpeed} \cdot \textit{Latency} + \\
&\qquad + \textit{shot-time} \cdot \textit{OurSpeed}) \\
\textit{shift-factor} &= \frac{\textit{TheirSpeed} \cdot \textit{Latency}}{\textit{TheirSpeed} \cdot \textit{Latency} + \textit{shot-time} \cdot \textit{OurSpeed}} \\
&= \frac{1}{1 + \frac{\textit{shot-time}}{\textit{Latency}} \frac{\textit{OurSpeed}}{\textit{TheirSpeed}}}.
\end{aligned}$$

```
Inputs:
  • stopping-location: the stopping location of the opponent generating a threat
Constants and helper functions:
  • Latency: an estimate of the total latency of the soccer team
  • KickSpeed: the maximum kick speed allowed by the rules (we assume that the opponent
    would shoot at this speed)
  • OurSpeed: an estimate of the top speed of our robots
  • TheirSpeed: an estimate of the top speed of our robots
  • OurGoal: the position of the center of our goal
Outputs:
  • secondary-location: the location for a secondary defender to guard against a shot from
    stopping-location
function SecondaryDefenseLocation(stopping-location)
    $\textit{shot-time} \leftarrow \frac{\textsc{Distance}(\textit{OurGoal}, \textit{stopping-location})}{\textit{KickSpeed}}$
    $\textit{shift-factor} \leftarrow (1 + (\textit{shot-time}/\textit{Latency})(\textit{OurSpeed}/\textit{TheirSpeed}))^{-1}$
    $\textit{secondary-location} \leftarrow \textit{stopping-location} + (\textit{OurGoal} - \textit{stopping-location}) \cdot \textit{shift-factor}$
    return secondary-location
end function
```

Algorithm 8: The algorithm to compute the position to place a secondary defender to block a shot threat.

5.6.2 Task positions for blocking passes

For blocking passes, we simply compute the midpoint of the segment between the threat location and the pass receiver location, as long as that point is outside our defense area. If it is, we instead choose the location that is on the segment and at a fixed minimum distance from the defense area.

5.7 Task assignment

Once the evaluator has generated the tasks, it needs to assign them to specific robots. Historically, this step was a difficult one to get right: at the first RoboCup tournament where we used this threat-based defense, one of the main problems was constant switching between different assignments, with the effect that defenders would end up stuck between tasks, with their targets constantly switching back and forth between two or more different locations. The switching was caused by inadequate hysteresis and an assignment function for secondary defenders that did not

For primary defenders, since they move around the defense area, we can reduce the problem to a one-dimensional one, as in Section 5.8: we sort the tasks that are destined for primary defenders by the angle of the vector from the center of our goal to each task's target location. Then we sort the defenders by the angle of the vector from the goal center to each robot's location, and assign the tasks to the robots in the same order.

For secondary defenders, we use the same optimal assignment engine that STP uses for assigning tactics to robots; here, we take the target location for each task and compute the navigation time for each robot to that location and use those values as the costs.

5.8 Three or more primary defenders

Corner kicks taken by the opponent present a challenging problem for any team's defense: the ball is close to the goal, which is dangerous in itself and means that teams tend to attack with more robots then than during the rest of the game. Defending well is also be made more difficult by the fact that the defenders need to place many robots within a small area around the goal. Assigning the appropriate defensive roles so that robots do not interfere with each other's navigation is particularly challenging when opponents quickly change formations. Figure 5.6 shows our solution to this, which involves assigning all of our defensive robots to be primary defenders.

This reduces the problem of positioning the defenders from a two-dimensional problem to a one-dimensional one, ensuring that the evaluator can assign positions to them in a manner which prevents collisions and interference. In this mode, the first-level threat and associated defender position or positions are computed as normal. For second-level threats, only shot-blocking tasks and only their positions near the defense are considered. All the positions are sorted by their linear position around the defense area. Now, in order to avoid collisions while allowing all robots to reach their target positions, the positions must be moved away from each other. The first-level position(s) are treated as fixed, since they represent the most important positions to block; the second-level positions are moved away from the first-level positions if necessary so that there is enough space between them for robots to reach the positions without colliding.

5.9 Defender behavior

In this section, we describe the way in which the defensive robots actually behave after the threat and task positions are computed. The tasks mostly assume (with snapping as a notable

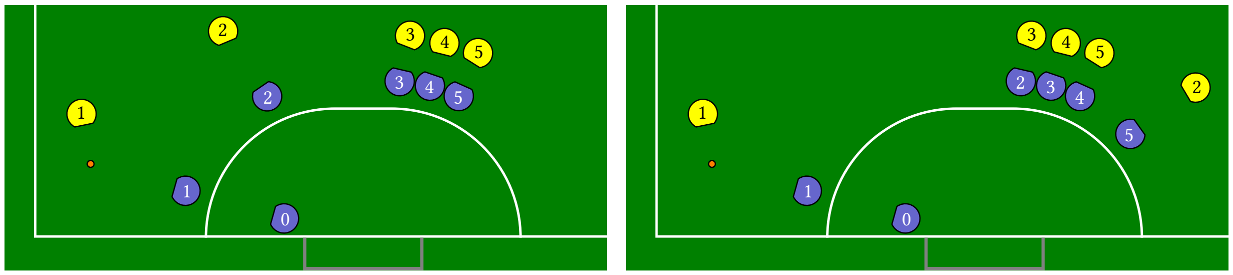


Figure 5.6: Examples of our defense's response to opponent positions while all defenders are primary defenders. If opponent 2 crosses behind the other three opponents, the defenders smoothly shift to follow the movement, staying in order without crossing past each other.

exception) that the world is static: given the positions of the opponents and the ball, we compute positions in response. That approach becomes inadequate for handling the full game, with dynamics brought in. As a result, we have added additional behaviors apart from simply moving to the computed positions.

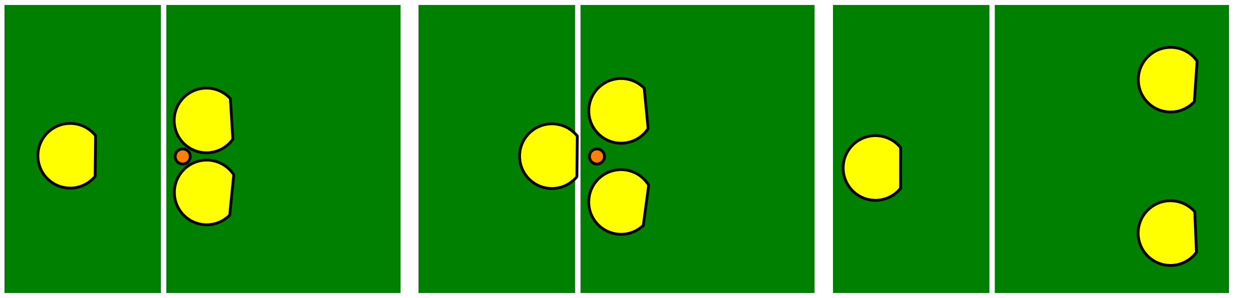
5.9.1 Closing the gap

As previously shown, when two robots are primary defenders, we leave a small gap between them, both to cover more of the open angle to the goal and to avoid causing mechanical problems that might arise from the robots trying to push against each other while driving around. While the goalie positions itself to block direct shots that pass between the defenders, bounces from the sides of the defenders may emerge from the gap at unpredictable angles. We have addressed this problem by forcing the primary defenders to drive toward each other as quickly as possible when there is an incoming shot aimed between them; this lets them form a solid wall in front of the ball, reducing the likelihood of unexpected bounces.

5.9.2 Guarding the ball

Since the defenders cannot touch the ball while they are even partially in the defense area, a ball that is stopped very close to our defense area (i.e., within one robot diameter of it) is difficult for them to deal with. The angles from which they can approach the ball are limited, and we are reluctant to have them point toward the goal while manipulating the ball, for fear of accidentally moving the ball toward our own goal. At the same time, such a ball represents an excellent opportunity for the opponents to score, so the defense must continue to carefully defend the ball.

To handle this situation, we closely coordinate the goalie and defenders so that the defenders can guard the ball while the goalie moves forward to kick it. We compute positions for the defenders that are as close as possible to the ball and each other at first, but allows them to move apart to leave room for the goalie as it drives forward to kick the ball according to its own normal behaviors. The combined movement of the goalie and defenders is demonstrated in Figure 5.7.



(a) The defenders close around the ball while the goalie moves forward. (b) The defenders part just enough to allow the goalie to reach the ball. (c) The goalie has kicked the ball and the defenders resume normal positioning for a distant ball.

Figure 5.7: An example of the coordinated guarding behavior of our defenders.

5.10 Goalie behavior

So far, we have discussed how the engine treats the defensive robots besides the goalie. In this section, we detail the behavior of the goalie, the very last line of defense before our goal. The goalie tactic already contained considerable complexity before the beginning of this thesis; here, we focus on the new behaviors added over the course of this thesis.

5.10.1 Corner kicks

As discussed in Section 5.8, corner kicks are a particularly hard case for many teams' defense strategies to handle. There, we discussed the defenders' behavior; here we go over the goalie's. A common strategy for teams to use during corner kicks is to use a chip kick to pass the ball to a teammate, positioned on the other side of the goal, who immediately kicks the ball at the goal. If the goalie follows its typical behavior of staying between the ball and the goal while the kick is being set up, this attack strategy forces it to move a long distance (all the way across the goal) to be able to cover the subsequent shot. To counteract this possibility, we alter the behavior of the goalie during corner kicks as follows. Before the opponent team takes the kick, the goalie remains on the line $y = 0$ (i.e., the line joining containing the center of the field and the centers of the goals), in order to avoid going too far to either side of the goal. In particular, it constantly moves to the intersection of that line with the line containing the locations of the opponent kicker and the ball, as long as that point is within a certain segment of the $y = 0$ line; if not, it moves to the point on the segment that is closest to the intersection. The segment we use extends from $\mathit{MaxRobotRadius}$ away from the goal to $\mathit{MaxRobotRadius} + .5 \cdot \mathit{DefenseRadius}$ from it. This behavior allows the goalie to react to the motions of the opponent kicker while reduce its ability to have a clear shot across the defense area while not moving too far from the center of the goal, either to the sides or toward the center of the field.

5.10.2 Diving

Our RRT-based navigation code is geared toward controlling robots such that they are stationary upon arriving at the target location. When the goalie is moving sideways to block an incoming shot, it is not important for it to be stationary; instead, it should make sure to get onto the path of the ball as quickly as possible. Ideally, it should also, if at all possible, have the ball collide with its kicker, so that it can immediately kick the ball away. For faster and more precise positioning in this case, we use the *diving* mode for the goalie, which directly computes an appropriate velocity command to send to the robot to intercept the ball as described. It decides whether to start diving based on the trajectory of the ball and our estimate of our robots' motion parameters.

The decision algorithm chooses to dive if all of the following conditions are true:

- the ball is coming toward the goal,
- the ball is moving fast enough to reach the goal if not stopped, and
- the ball is close enough and the goalie's acceleration and top speed are not high enough that normal navigation would allow it to intercept the ball with sufficient time to spare.

If the conditions are all true, then the diving algorithm attempts to maneuver the goalie so that it moves on a line perpendicular to the projected trajectory of the ball, crossing the trajectory at the same time as the ball reaches the same point. Each timestep, it computes the average speed that would result in arriving at the right time. If its current speed is lower, the computed speed is the current speed plus the robot's maximum acceleration times a quantity we call the *translational lookahead.* If its current speed is higher than the desired speed, the computed speed is the current speed minus the robot's maximum deceleration times the lookahead. The translational lookahead is a time interval tuned to reflect the amount of time the robot takes to respond to commands; adjusting the computed velocity values to account for it prevents the robot from overshooting or undershooting.

5.10.3 Kick taking

The RoboCup 2016 tournament introduced a new rule which prohibited players from entering either team's defense area; in previous years, players could enter the opposing team's defense area, as long as they did not touch the opponent goalie. As a result, if a team's goalie has control of the ball inside its own team's defense area, it is completely safe from interference by the other team. We observed that this is effectively the chance to turn a shot on our goal into a free kick; to take advantage of this opportunity, we added the ability for the goalie to behave as a free kick taker under appropriate conditions. Once it begins a kick, it will continue as long as the ball is within the defense area.

To make these conditions more likely to be satisfied in the course of a game, we modified the normal clearing behavior to not enable the goalie's kicker by default. Doing so allows the goalie to stop a ball and gain control of it, instead of always kicking it away. Once the conditions have been met, the goalie begins to set up a kick.

The modularization of our code into tactics makes it possible for the goalie tactic to simply call the tactic that takes actual free kicks; the kick taker tactic will automatically handle positioning and coordination with the receiver tactics. The kick taker tactic is careful to avoid touching the ball before it is ready to kick the ball, since that would result in a foul for a normal free kick; that property also makes it suitable for handling the ball even when it is near the goal, as we need to be careful not to accidentally push the ball closer to the goal.

5.11 Defense visualization

Across all the games that CMDragons played at RoboCup 2015, we scored a total of 48 goals, while no goals were scored against us — a strong real-world demonstration of the effectiveness of the defense. In this section, we demonstrate the application of our visualizations to the defense domain by describing some of the defense-related situations that occurred during these games.

5.11.1 Open angles

The concept of open angles to our goal is critical to the defense, so we always have a drawing depicting the current open angles, as shown in Figure 5.8. The defense computes open angles with only the primary defenders robots as obstacles, since it cannot count on attacking robots, much less opponents, to move suitably for defense, and secondary defenders are more likely to have to move quickly and suddenly. Figure 5.8 shows visualizations showing the open angles for one moment in time; visible are two very narrow open angles at the edges of the goals, shown in red (each visible line is actually two lines very close together), as well as a wider one outlined in blue. We distinguish the angle between the primary defenders because the goalie prefers to position itself there, avoiding moving too far toward either corner of the goal.

5.11.2 Projected ball trajectory

As discussed in Section 5.3.3, the defense evaluator can override the default Kalman filter estimate of the ball position and velocity in cases where that original estimate is misleading. It then maps the results of the calculation into drawings as follows, and as shown in Figure 5.9. To depict the new state, it draws a line (purple) that starts at the position and ends at the position plus a constant multiplied by the velocity. To depict the original state, it draws a pair of lines (orange), which are computed similarly to the first line based on the original position and velocity, but then shifted a short distance perpendicular to the velocity. Drawing the lines in this way produces the effect that the lines for the original and new states do not overlap when the states closely agree, without introducing an asymmetric sideways shift in one direction. In the moment shown in the figure, an opponent has just kicked the ball toward our goal; the magnitude of the Kalman filter's velocity estimate has not increased to the full correct value yet, while the defense estimate's magnitude has, meaning that the defense has more time to react to the shot.

5.12 Discussion and summary

In this chapter, we introduced a sophisticated threat-based defense algorithm for our SSL team. We have refined the algorithm over four consecutive RoboCup SSL tournaments, and we

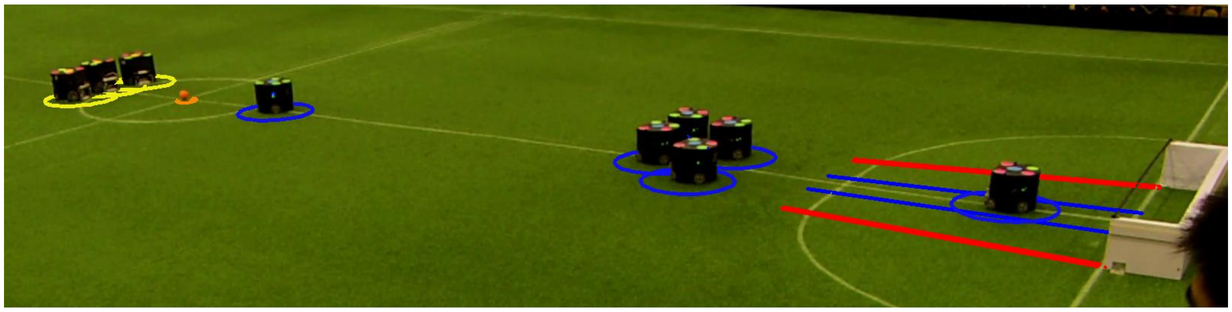


Figure 5.8: An example of the visualizations showing the open angles from the ball to our goal.

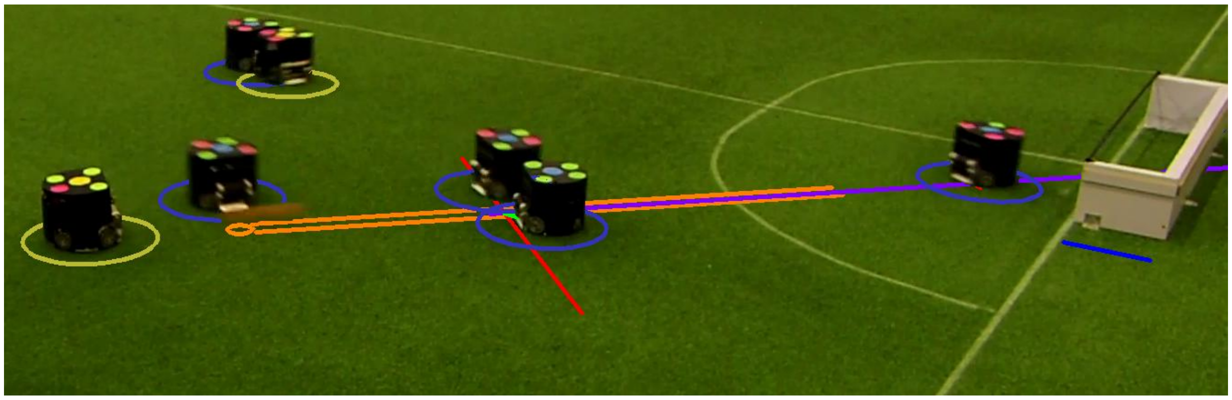


Figure 5.9: An example of the visualizations generated by the defense-specific ball velocity estimator when the defense evaluator is using it to override the Kalman filter.

particularly demonstrated the strength of its performance at RoboCup 2015, where CMDragons scored 48 goals and conceded none[1]. The defense algorithm handles in a general manner the different ways that the opponent team may score on our goal and chooses the most important threats to defend against, then intelligently assigns the available defender robots to the tasks.

The defense algorithm, as we discussed, contains not only the core threat evaluation and task assignment algorithm, but also specific behaviors for the defending robots. We have the goalie, which stays nearest to the goal, primary defenders, which move just outside the defense area, and secondary defenders, which move farther afield. Each one has additional robot-specific behaviors on top of the generic tasks created by the central evaluator, acting to improve the response of the defense within the constraint of covering the threat for that task. Taken together, the evaluator and robot behaviors interact to produce a coherent defense that has served us well under a wide variety of real-world scenarios, some of which we demonstrated using our augmented reality visualizations.

[1]We acknowledge that the concurrent developments in the team's offense may also have contributed somewhat to this result.

Chapter 6

RoboCup SSL: Autoref

In this chapter, we move from discussing the SSL to discussing our autoref for it. In Section 6.1, we discuss how referees and autorefs fit into the SSL ecosystem. In Section 6.2 and Section 6.3, we describe a formalization, as a hybrid automaton, of the structure of an SSL game (and hence of any referee's internal model of the game), and then move on to our implementation of a means of updating the model according to external input. Section 6.4 describes our observations and results from using the autoref, and Section 6.5 details how we apply the overarching theme of visualization to the autoref.

Here, the flavor of the algorithm and visualizations is a little different from those of the quadrotor and SSL domains, as a result of the nature of the autoref as compared to the robots discussed previously. Rather than describing future plans, the autoref and the information produced by it are more focused on the past; the autoref's decisions are based on the history of world states, but the extent of its future planning is minimal. The autoref is an autonomous agent all the same, and it influences the state of the world, but via purely informational actions, rather than physical ones. In a way, this property makes it even better-suited as an application domain for our visualizations: the effects of its decisions are visible to observers only indirectly through the response of the teams, unlike the actions of the teams themselves. (We have used a dedicated computer display for the autoref, but even then the direct output of the autoref does not involve motion of physical objects.)

6.1 The referee interface: inputs and outputs

The goal of an autoref is to algorithmically replace the human referees. In order to produce a system to achieve that goal, it helps to understand the external interface that an autoref — or, in some sense, a human referee operating a refbox — presents to the other SSL programs on the network.

Viewed from the outside, the autoref receives updates from SSL-Vision in the same way that the teams themselves do, interprets them according to the rules of the game, and outputs commands back to the teams at the appropriate times.

Accordingly, sixty times per second, the autoref receives as input the values in Table 6.2, which are collectively referred to as a *world state*; we denote by $\mathcal{W}$ the set of all possible world

Name	Description
`HALT`	all robots must stop moving immediately; game time stops counting
`STOP`	play temporarily stops to set up a kick; robots must move slowly and stay away from the ball
`INDIRECT(TEAM)`	`TEAM` must take an indirect free kick within 10 seconds
`DIRECT(TEAM)`	`TEAM` must take a direct free kick within 10 seconds
`KICKOFF(TEAM)`	`TEAM` should prepare to take a kickoff
`GOAL(TEAM)`	a goal has just been scored by `TEAM`; also, teams must behave as if the `STOP` command had been given
`READY`	given after `KICKOFF_TEAM`; `TEAM` must take kickoff within 10 seconds
`START`	play starts immediately and any robot may touch the ball

Table 6.1: The referee commands used in the SSL.

states.

The time is given in seconds since the Unix epoch, and all physical coordinates are given in millimeters, in a coordinate system with its origin at the center of the field, positive x-axis pointing toward a goal, and positive y-axis pointing 90° counterclockwise of the positive x-axis.

Name	Description
t	the current time
n	the number of robots on the field
T^i	team of the i^{th} robot (`BLUE` or `YELLOW`)
$\vec{p}_r^i$	position of the i^{th} robot
$\vec{p}_b$	position of the ball

Table 6.2: The values used as input by the autoref.

As output, an autoref provides two forms of primitives, corresponding to the actions of the human referees: referee commands and ball reset positions. The referee commands are the same as those transmitted by the refbox used by the human referees. The ball reset positions are the replacement for a human referee's ability to physically place the ball on the field: whenever a human referee would be placing the ball somewhere on the field, the autoref outputs a position, and assumes that some means exists for the commands to be interpreted and carried out. This means may consist of, for example, a human watching the output of the autoref, a specialized robot, or, with the cooperation of the teams, the players themselves.

The definition of this interface allows an autoref system like the one we describe here to interact with other structurally similar games, not only the SSL. With the appropriate vision input, translation of commands and coordinates, the system could conceivably be used to referee, for example, a game of the MSL instead, using the SSL rules. Of course, the autoref relies upon global ground truth data on the positions of the robots and ball, which would not be generally available in the MSL without additional equipment.

6.2 Hybrid automaton

We formalized the structure of an SSL game by taking the natural-language rules document of the SSL and creating a hybrid automaton that describes the structure of the game. A hybrid automaton is a means of representing a system which contains both discrete and continuous elements that can change over time. We give a brief overview of the hybrid automaton formalism in general [Alur et al., 1993], and then of the automaton which specifically describes a game in the RoboCup SSL.

6.2.1 Hybrid automaton formalism

In its most general form, a hybrid automaton consists of the following components [Henzinger, 2000]:

- V_D: A finite set of real-valued *variables*. A *valuation* is a map from each variable to a value, and Σ_D is the set of all valuations.
- Q: A finite set of *locations*.
- μ_1: A map from Q to sets of *activities*, which are C^∞ functions from $\mathbb{R}^{\geq 0}$, the set of nonnegative real numbers, to Σ_D.
- μ_2: A map from Q to *exception sets*, which are subsets of Σ_D.
- μ_3: A map from Q^2 to *transition relations*, which are subsets of Σ_D^2. We describe $\mu_3(q_1, q_2)$ as the transition from q_1 to q_2.

For an interval I in $\mathbb{R}^{\geq 0}$, denote its left and right endpoints by l_I and r_I respectively. An interval may be either closed or open at each of its endpoints.

A *trajectory* of a hybrid system is a sequence of tuples $(\sigma_i, \ell_i, I_i, f_i, \sigma'_i)$ such that

- For every $i \geq 0$, ℓ_i is a location, σ_i is a valuation, and $(\ell_{i+1}, \sigma_{i+1})$ is a successor of (ℓ'_i, σ'_i); that is, $(\sigma'_i, \sigma_{i+1}) \in \mu_3(\ell_i, \ell_{i+1})$.
- The I_i are intervals that partition $\mathbb{R}^{\geq 0}$ and are in increasing order by index.
- For every $i \geq 0$, (a) $f_i \in \mu_1(\ell_i)$, (b) $f_i(0) = \sigma_i$, (c) $f_i(r_{I_i} - l_{I_i}) = \sigma'_i$, and (d) for all $t \in I_i$, $f_i(t - l_{I_i}) \notin \mu_2(\ell_i)$.

In effect, a trajectory describes a possible evolution of the automaton over time, where the state may evolve continuously according to the trajectories or discretely according to the transition relations. The exception sets define conditions which force the automaton to undergo a transition out of a location. A common scheme, which we will use here, is to express the sets of activities as the solution sets of differential equations. Additionally, an automaton may effectively include discrete variables if those variables are constant in all allowed activities and change only in transitions.

Hybrid automata are mostly used in the context of formal verification, and so authors are typically concerned with characterizing the set of all possible trajectories of an automaton over time. Instead, we think of and describe an automaton as a machine which follows a specific trajectory in any given instantiation.

Accordingly, we may replace the transition relations with *transition functions*, so that the range of μ_3 is instead the partial functions $\Sigma_D \to \Sigma_D$. Then we are concerned with trajectories such that for each i, $\sigma_{i+1} = \mu_3(\ell_i, \ell_{i+1})(\sigma_i')$.

We also define the exception sets implicitly by associating a guard condition (a predicate over valuations) with each transition; the exception set for each location will then consist of all valuations which satisfy the guard condition for some transition, and the domain of each transition function will be the set of valuations satisfying the associated guard condition. This forces the automaton to transition as soon as the current valuation satisfies the guard condition for some transition.

6.2.2 SSL automaton

For the automaton describing an SSL game specifically, the locations, which are described in Table 6.3, capture the idea that, roughly, the game is either running or stopped, and each of those entails a distinct set of conditions that are relevant to refereeing. The auxiliary variables required to represent the state of the game are described in Table 6.6, and the differential equations describing the continuous evolution of the variables is described in Table 6.4. One of the variables, the current stage, describes what overall part of the game is currently happening: either one of the halves, halftime, or the time before or after the whole game; these values are described in Table 6.5.

More explicitly, the components of the model listed above correspond to the following real-world quantities relating to the autoref:

- V_D is a set of variables describing the state of the world and the game (e.g. time left in the half, ball x position, etc.).
- Σ_D is the set of values of the variables, which we call $\mathcal{W}$.
- Q is the set of high-level states of the game, as described in Section 4.6.
- μ_1 describes the possible ways for the world state to evolve within each high-level game state, according to the laws of the game and the laws of physics.
- μ_2 describes the sets of world states within each high-level state that would cause the high-level state to change to any different one (e.g., because the ball left the field or a player violated a rule).
- μ_3 describes the individual conditions that cause the high-level state to transition between each possible pair of values.

We describe the occurrence of a transition in the automaton as an *event*; the continuous evolution of the automaton is interrupted whenever an event occurs. For the implementation, events are triggered by detector objects that examine the current world state and check whether any relevant rule applies to the current situation.

We also define some notation and functions operating on the values of these variables, for

Location	Description
GAME_ON	The game is actively being played. This location typically ends when a robot commits a foul or the ball leaves the field.
GAME_OFF	The game has been stopped for one of the preceding reasons, and the robots must not come near the ball while the referee is positioning it. This location ends once the robots are ready to kick, which is usually taken to be when they stop moving or after some time has passed.
SETUP	The command to take a kick has been given, but the kick has not been taken yet. This location ends once the kick is taken, or too much time has passed without a kick.
BREAK	One team has called a timeout or the game is in a break between periods. This location only ends if the team ends the timeout or time in the break period runs out.

Table 6.3: The high-level automaton locations of an SSL game.

Locations	Equations
all	$\frac{d}{dt}\text{time} = 1$
BREAK	$\frac{d}{dt}\text{timeout-time(timeout-team)} = -1$
all but BREAK	$\frac{d}{dt}x = -1$ for all $x \in \text{y-times(BLUE)} \cup \text{y-times(YELLOW)}$

Table 6.4: The differential equations for the continuous variables of an SSL game; see Table 6.6 for descriptions of the variables. Most of the continuous evolution of the game is under the direct control of the team agents, rather than the referee, so these equations are minimal. By a slight abuse of notation, the third equation indicates that all the elements of y-times(BLUE) and y-times(YELLOW) have derivative −1.

Name	Description
PRE-FIRST	the READY command for the first kickoff of the first half has hasn't been given yet
FIRST	the initial READY command for the first half has been given, and the first half hasn't ended
HALFTIME	the first half has ended and the robots are halted
PRE-SECOND	the first half has ended and the initial READY command of the second half hasn't been given yet
SECOND	like FIRST, but for the second half
POST	the second half has ended

Table 6.5: The coarse stages of an SSL game.

Name	Domain	Description
time	$\mathbb{R}$ (seconds)	the current time
stage-end	$\mathbb{R}$ (seconds)	the time at which the current stage will end
timeout-time(T)	$\mathbb{R}_{\geq 0}$ (seconds)	remaining timeout time for team T
timeout-num(T)	$\mathbb{N}$	the number of remaining times that team T may call a timeout
reset-position	$\mathbb{R}^2$ (meters)	the position where the ball should be placed for the next kick
kick-deadline	$\mathbb{R}$ (seconds)	the time at which the game will be reset to a forced start if the current kick is not taken yet
call	referee commands (Table 6.1)	the command that should be transmitted next (used while the game is stopped)
stage	game stages (Table 6.5)	the current coarse stage of the game
touch-team	{blue, yellow}	the last team to touch the ball while the game was running
kick-team	{blue, yellow}	the team currently permitted to kick the ball
goal-allowed	{true, false}	whether a direct goal is currently allowed (true except immediately after indirect free kicks)
kicker	{robots on the field} $\cup$ $\{\mathit{null}\}$	the robot which just took a free kick
r-cards(T)	$\mathbb{N}$	number of red cards for team T
y-times(T)	lists of elements of $R_{\geq 0}$	time remaining for each yellow card for team T

Table 6.6: The auxiliary variables of the automaton representing an SSL game.

convenience.

- team(R): the team of robot R
- stage-end-cmd(s): the command that is given to end stage s (KICKOFF(T) for halftime, HALT for game halves)
- stage-length(s): the length in seconds of stage s
- next-stage(s): the stage that comes after stage s
- !T: the opponents of team T

In this presentation, most of the predicates are described in plain text, with the details of how to receive inputs and evaluate them left abstracted away. It should be possible to incorporate a more detailed model of their computation into the automaton without drastic changes, by adding some additional variables and self-loops. Similarly, the change over time of the variables describing the world state can be considered to evolve according to complex equations which are part of the automaton but unspecified. It may also make sense to actually model the event detection as a separate automaton; the hybrid automaton model allows for the composition of two automata, which can communicate by emitting and receiving events, and the generation of the primitive events can be thought of as coming from another unspecified automaton.

Figure 6.1 shows a simplified representation of the automaton, with only the set of locations and which pairs have edges between them; each edge there corresponds to multiple possible transitions, with the transitions between each pair of locations given a high-level description of what kinds of rules form it. The details of the transitions are specified in the appendix. For simplicity, we have omitted the detailed rules for penalty kicks and penalty shootouts, as well as the calculations for kick reset positions. (A detailed example of the position calculation is given in Algorithm 10 in Section 6.3.2.) In each table, the first column represents the preconditions on the world state for the transition to occur, the second represents the referee command component of the action that is transmitted when the transition occurs, and the third column represents the changes made to the internal state variables.

The overall locations correspond to those in mentioned in Section 4.6. When the game is in the GAME_ON location, the ball is in play and the robots are actively trying to score goals; during GAME_OFF, the ball is being placed for the next restart of play; during SETUP, the ball has been placed and the robots may set up for a kick.

We have described this automaton as abstractly representing the structure and state of a game, as defined by the rulebook, but there is a direct correspondence to what human referees and autorefs must do while they observe the game. They are also essentially modeling the same automaton: they must use their observations to estimate when the true state of the game ought to change, so that they can issue the appropriate commands.

6.2.3 Automaton evolution and the rules

In this section, we discuss some examples of how the changes in game state combine with the rules of the game to produce the transitions that the automaton encodes.

Imagine that the game is actively being played, but then the ball goes out after being touched by the blue team; play must stop so that the referee may retrieve and place the ball, then signal

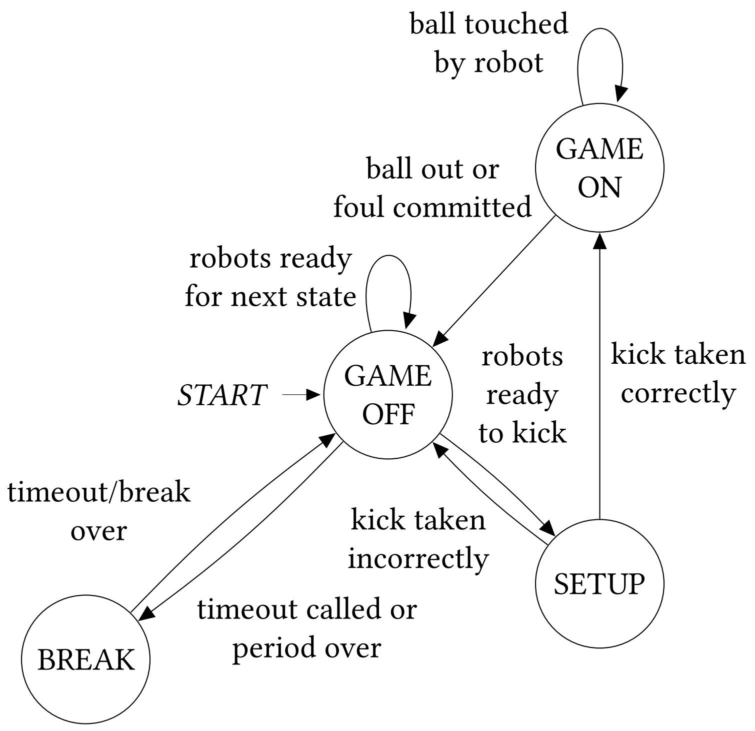


Figure 6.1: A simplified representation of the automaton representing a game of the SSL.

the next kick. However, suppose the team has a software problem and does not take the kick before 10 seconds have passed, so the referee stops the game again and then calls for a neutral restart. In order, the individual components of the changes in the automaton state are as follows:

- The location is GAME_ON.
- The blue team touches the ball. The touch event sets the touch-team variable to BLUE.
- The ball goes out. The STOP command is transmitted, which tells teams that the game is stopped and that they should prepare for some kick to come next; internally, the location changes to GAME_OFF, call is set to INDIRECT, and kick-team is set to `YELLOW`.
- Once the ball is positioned and the robots settle, the `INDIRECT(YELLOW)` command is transmitted, which tells the yellow team that it is to take a kick within 10 seconds. Accordingly, the location changes to SETUP and kick-deadline is set to time + 10.
- Once time becomes greater than kick-deadline (i.e., 10 seconds pass and the team does not take the kick), the `STOP` command is transmitted, since there is about to be a new restart of the game; the location changes back to GAME_OFF and call is set to `START`.
- Once the robots settle again, the `START` command is transmitted to indicate that gameplay has resumed and either team may touch the ball, and the location changes to GAME_ON.

As another example, consider the sequence of events that occurs when a goal is scored by the yellow team. According to the definition of the commands as given by the SSL, after the game is stopped there must be a `KICKOFF` command for the appropriate team, which is then followed by a `READY` command, and only then is the team allowed to actually kick the ball — as opposed to a free kick, where only a single command is given after the game is stopped.

- The location is GAME_ON.
- The ball enters the blue goal, so the `GOAL(YELLOW)` command is transmitted to indicate to the teams that the yellow team has scored. The game must now stop to set up for a kickoff by the blue team, so the location changes to GAME_OFF, call is set to `KICKOFF`, and kick-team is set to BLUE.
- Once the ball is positioned and the robots settle, the transition from GAME_OFF to itself is taken; the `KICKOFF(BLUE)` command is transmitted, but the location remains equal to GAME_OFF.
- Once the robots settle again, now knowing that it will be a kickoff for the blue team, the location becomes SETUP and the `READY` command is transmitted, indicating that the blue team may take the kick now.
- Once the blue team takes the kick, gameplay has started again, so the location becomes GAME_ON.

Figure 6.2 graphically demonstrates the sequence of transitions that occurs at the beginning of a typical game, with a kickoff given, a brief period of play, and then a ball exit leading to an indirect free kick.

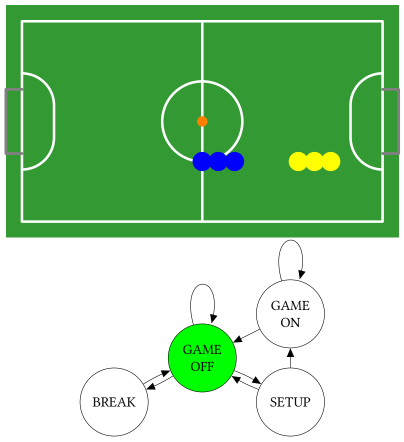
Finally, we briefly discuss the use of the kicker and goal-allowed variables, which are used for two intertwined rules. First, if a robot takes a free kick and is the first robot to touch the ball afterward, play stops and the other team receives an indirect free kick. Second, a goal may not be scored directly from an indirect free kick; that is, a goal is not allowed if the current episode was started by an indirect free kick and the last robot to touch the ball is the robot that took the kick. (If the ball enters a goal, it is treated as simply having left the field over the goal line.) Accordingly, when an indirect free kick is signaled, goal-allowed is set to false; for any other kick, it is set to true. When a kick is actually taken, kicker is set to the robot that took it. As long as the ball is in play and no robots touch it, neither variable changes. When a robot touches the ball, what happens next depends on whether that robot is the same as kicker. If it is, this is a double touch and the game stops. Otherwise, kicker is set to *null* (i.e., some sentinel value that is equal to no robot), since a double touch is no longer possible. Simultaneously, goal-allowed is set to true, since a second touch means a goal is now allowed.

6.3 Automaton events

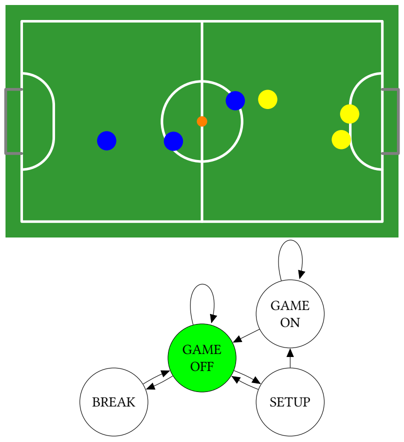
In this section, we describe our concrete algorithm and implementation of the events described previously. To model the evolution of the automaton over time, the algorithm handles world updates in a loop, taking the possible conditions that trigger transitions within the automaton and estimating whether the conditions for each one have been satisfied.

6.3.1 Event loop algorithm

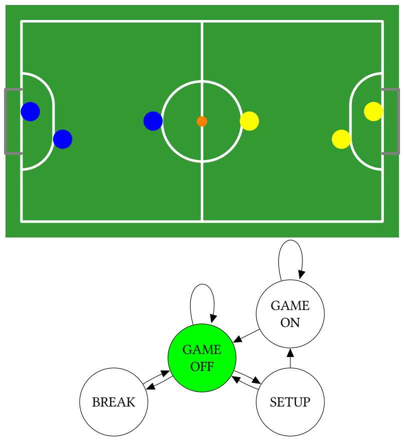
We implemented a program to approximately track the evolution of the automaton described above in terms of a set of separate *event detectors*, each of which receives the sequence of world



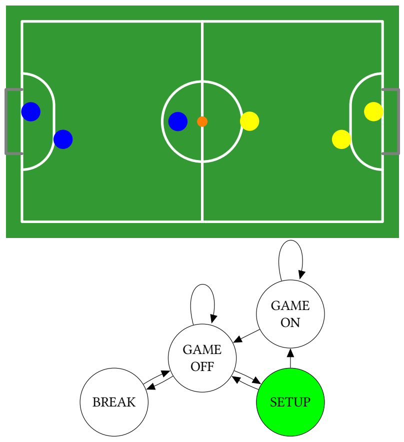
(a) The robots are in their starting positions when the autoref first transmits the STOP command.



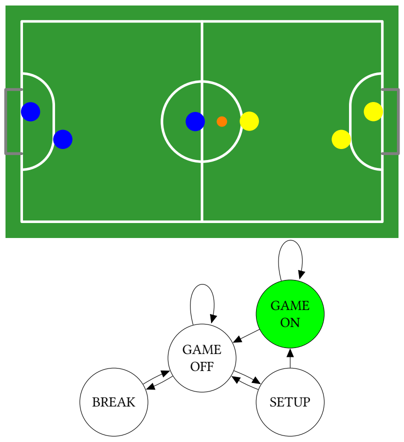
(b) The robots have begun moving to new positions in response to the STOP command.



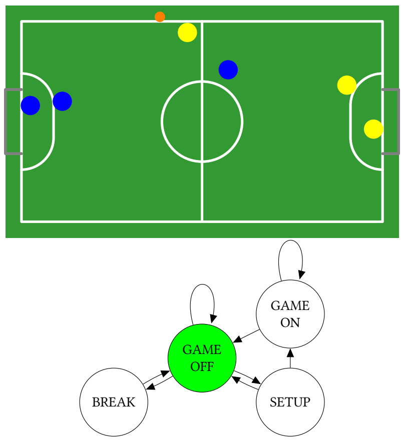
(c) The robots have settled, so the autoref now transmits KICKOFF and waits for them to settle again.



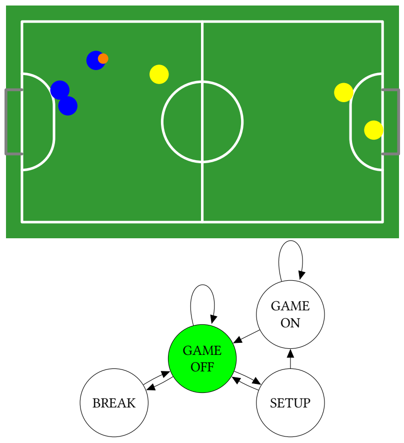
(d) The robots have settled again; the autoref now transmits READY, so the blue team may take the kick.



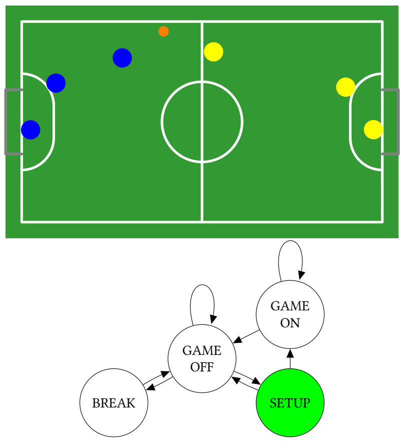
(e) The blue robot has just taken the kick, so the location becomes GAME_ON.



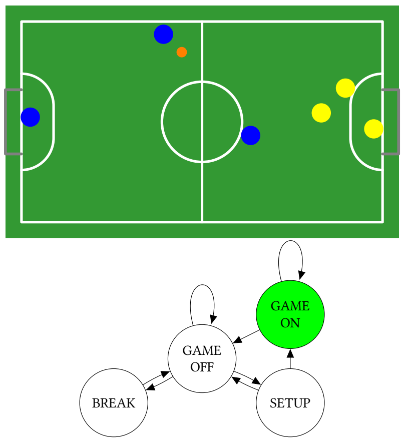
(f) The ball has left the field; the location becomes GAME_OFF so the ball may be positioned for a free kick.



(g) The referee robot (blue goalie) pushes the ball to its reset location.



(h) The robots have settled, so the blue team is free to take the kick.



(i) The blue team takes the kick and the location returns to GAME_ON.

Figure 6.2: A demonstration of the evolution of the autoref over time in response to the motions of the robots and ball. Each subfigure shows the field at some time, with the corresponding automaton location highlighted below.

states and outputs an object describing how the state of the automaton should be updated. The algorithm implements a sort of discrete approximation of the true continuous-time automaton, since it can only receive discrete updates of the world state. It operates at the full frame rate of SSL-Vision.

The architecture described here is based on that of CMCast [Veloso et al., 2008]; however, we emphasize that the overall algorithm can implement a general hybrid automaton, with a suitable set of event detectors plugged in.

In this kind of architecture, an event is any of the conditions associated with an edge in the automaton. The events we have currently implemented are:

- the ball's speed goes above the maximum limit defined in the rules,
- the ball enters a goal,
- the ball exits the field without entering a goal,
- the robots are ready for a free kick to be taken,
- a robot touches the ball,
- a robot takes a free kick,
- a robot dribbles the ball over a linear distance of greater than 1000 mm,
- a team is taking too long to take a free kick,
- progress gets stuck during play, and
- the end of the first or second half is reached.

Each time a new world state is available, the autoref algorithm checks whether any event has occurred; if so, it updates its state variables and may issue a command accordingly.

The state and values of the auxiliary variables of the automaton are described by $v \in \mathcal{V}$. We also make use of a set of *action* objects, each representing an externally visible action that the program can take. The contents of $\mathcal{A}$ may vary depending on the application. For the RoboCup SSL, any particular action consists of the outputs described in Section 6.1: signaling that the ball should be positioned at a particular point or transmitting a new referee command to teams (and possibly both or neither).

In general, an event may conceptually be thought of as depending on the full history of world states; however, we expect that events in practice will not need to truly operate on the whole history, but can instead rely on a small fixed set of variables calculated from each new world state and the previous values of those variables. For the purpose of fitting within the hybrid automaton framework, it is also required to limit the system to a fixed number of variables. Therefore, with each event e we associate a set $\mathcal{S}_e$ which contains possible "summaries" of the world state as relevant to the event. Then, we can conceptually think of the event as a function of type $\mathcal{W} \times \mathcal{S}_e \times \mathcal{V} \to (\mathcal{S}_e \times \mathcal{V} \times \mathcal{A}) \cup \{null\}$: each time a new world state is received, the function is provided with that state, its own summary of previous world states, and the current internal variables, and it returns either *null*, indicating that the event did not fire, or the new summary, variables, and an action to take. In practice, to more closely mirror the structure of the actual code, we will work with each $\mathcal{S}_e$ by treating it as a collection of variables, much like the autoref auxiliary variables, and write the changes to the summary variables as imperative assignments within the pseudocode.

The main action selection algorithm is given in Algorithm 9, where $\mathcal{E}$ denotes the list of relevant events and s_e denotes the summary for an event $e \in \mathcal{E}$. This algorithm first updates the variables according to the differential equations of the current location (denoted by the call to ContinuousUpdate), then handles the discrete transitions. It finds the first event that is firing, executes the action, and updates its variables; then it returns the beginning of the list and repeats until no events fire. This procedure allows multiple transitions to occur due to the same input, which is occasionally necessary, as discussed by Mosterman [1999]. Since the continuously updated global variables for the autoref are particularly simple, we will not further discuss the continuous update.

```
Inputs:
  • $w$: world state
Constants and helper functions:
  • $\mathcal{E}$: the set of event detectors
function RunEvents($w$, $v$)
    ContinuousUpdate($w$, $v$)
    $fired \leftarrow True$
    while $fired$ do
        $fired \leftarrow False$
        for $e \in \mathcal{E}$ do
            $ret \leftarrow e(w, v)$
            if $ret \neq Null$ then
                $v, a \leftarrow ret$
                $fired \leftarrow True$
                execute $a$
                break
            end if
        end for
    end while
end function
```

Algorithm 9: The overall event-handling loop algorithm.

6.3.2 Event detector implementations

Now we provide concrete examples of the implementations of events to demonstrate the detection algorithms and how the event loop corresponds to the hybrid automaton. Note that for this specific automaton, the only variables which update continuously are the current time and, during a team's timeout, the amount of timeout time remaining to that team.

In the pseudocode listings in this section, v and a refer to elements of $\mathcal{V}$ and $\mathcal{A}$. Construction of v and a are expressed by assignments to bracketed components of them, corresponding to the variables described in Table 6.6 and actions described in Section 6.1:

- $v[call]$: the command that should be transmitted next.

- $v[loc]$: the new location of the referee automaton.
- $v[touch\text{-}team]$: the last team to touch the ball.
- $v[touch\text{-}time]$: the last team to touch the ball.
- $v[touch\text{-}pos]$: the last team to touch the ball.
- $a[cmd]$: the command to be sent to the teams immediately.
- $a[reset]$: the point at which to place the ball for the next kick.

For components of the event-specific state summary, we use normal variables; assignments before the function bodies indicate their initial values before the first execution of the event.

Ball exit

Most episodes end with the ball exiting the field, making it one of the most common events in a game. Algorithm 10 shows a simplified version of the code used to detect when the ball has left the field and determine the appropriate response.

Lines 4–13 determine whether the ball exit event should be considered to be happening in the current frame. It uses s_{last} as a counter, so that it only makes the call if the observed position of the ball is outside the field for 3 consecutive frames, in order to reduce the incidence of false positives due to incorrect data. The remaining lines determine the appropriate response if the event is happening. Lines 14–15 indicate that, in any case, the game should be stopped; Line 16 assigns the kick to the appropriate team. Lines 17–26 compute where the ball should be placed for the subsequent free kick, according to whether the ball passed over the touch line and should be a throw-in, or passed over the goal line and, depending on which team touched it last, should be a goal kick or corner kick. Finally, Line 27 creates an appropriate visualization for this event and Line 28 returns the updated autoref variables from the event.

Ball stuck

If the teams appear to have reached a state such that, in the referee's judgment, the game is unlikely to continue making substantive progress, the referee may stop the game and call for a forced restart. We call this event a "ball stuck" event. We implement this event by periodically checking whether the location of the ball is within some fixed distance of its location at the last check; if it so sufficiently many times in a row, we consider the ball stuck. Algorithm 11 describes this algorithm.

Ball touching

Detecting when a robot has touched the ball is crucial to accurate refereeing, and is one of the most difficult individual events to detect.

We started off with the touch detector that was already present in the CMDragons codebase, which we will call the Accel algorithm. Each frame, it estimates the acceleration of the ball as follows. Let $\vec{p}_b^0$ be the most recently observed ball position and $\vec{p}_b^1$ and $\vec{p}_b^2$ be the two before that, in order. The algorithm estimates the acceleration of the ball by computing

Inputs:
- w: world state
- v: initial values of the autoref variables

Outputs:
- v: updated values of the autoref variables
- a: action to take if this event fires

Constants and helper functions:
- CrossBoundaryPoint(u, v): the point on the line segment between the positions of u and v that lies on the field boundary
- OnTouchLine(u): whether the point u lies on a touch line of the field
- DefenseTeam(s): the team that is defending the goal whose x-coordinate has sign s

```
1:  $\textit{exit-count} \leftarrow 0$; $\textit{last-ball} \leftarrow \textit{Null}$
2:  function BallExitEvent($w$, $v$)
3:      $a \leftarrow$ new action
4:      if $\textit{last-ball} = \textit{Null} \lor \textsc{InsideField}(\textit{ball})$ then
5:          $\textit{last-ball} \leftarrow \textit{ball}$
6:          $\textit{exit-count} \leftarrow 0$
7:          return $\textit{Null}$
8:      else
9:          $\textit{exit-count} \leftarrow \textit{exit-count} + 1$
10:     end if
11:     if $\textit{exit-count} < 3$ then
12:         return $\textit{Null}$
13:     end if
14:     $a[\textit{cmd}] \leftarrow \texttt{STOP}$
15:     $v[\textit{loc}] \leftarrow$ GAME_OFF
16:     $v[\textit{call}] \leftarrow \texttt{INDIRECT}(!v[\textit{touch-team}])$
17:     $\langle c_x, c_y \rangle \leftarrow \textsc{CrossBoundaryPoint}(\textit{last-ball}, \textit{ball})$
18:     if $\textsc{OnTouchLine}(c)$ then
19:         $a[\textit{reset}] \leftarrow \langle c_x, c_y - 100\,\text{mm} \cdot \textsc{Sgn}(c_y) \rangle$
20:     else
21:         if $v[\textit{touch-team}] = \textsc{DefenseTeam}(\textsc{Sgn}(c_x))$ then
22:             $a[\textit{reset}] \leftarrow \langle \textsc{Sgn}(c_x) \cdot (\textit{FieldLengthH} - 100\,\text{mm}), \textsc{Sgn}(c_y) \cdot (\textit{FieldWidthH} - 100\,\text{mm}) \rangle$
23:         else
24:             $a[\textit{reset}] \leftarrow \langle \textsc{Sgn}(c_x) \cdot (\textit{FieldLengthH} - 500\,\text{mm}), \textsc{Sgn}(c_y) \cdot (\textit{FieldWidthH} - 100\,\text{mm}) \rangle$
25:         end if
26:     end if
27:     $\textsc{Output}(\{\{\textit{touch-time}; t\}; \{\{\{\text{line}; \text{Pos}(\textit{ball}); \textit{last-touch}\}; \textit{ColorWhite}; \textit{BallStuck}; 0\}\}\})$
28:     return $v$, $a$
29: end function
```

Algorithm 10: The algorithm for determining whether a the ball has left the field and how to respond if so. Note that, in this algorithm and the following ones, we symbolically include the world state w as an argument to the function to avoid clutter, but concretely use values such as *ball* and t that are part of that world state.

Inputs:

- *w*: world state
- *v*: initial values of the autoref variables

Outputs:

- *v*: updated values of the autoref variables
- *a*: action to take if this event fires

Constants and helper functions:

- *CheckInterval* = 1 s: the time between checks of the ball's location
- *StuckThreshold* = 250 mm: the maximum distance between the ball's locations on two consecutive checks for the second check to contribute toward considering the ball stuck
- *StuckCount* = 12: the minimum number of consecutive checks during which the ball has moved at most *StuckThreshold* to count the ball as stuck

```
$last\text{-}check\text{-}time \leftarrow -\infty$
$stuck\text{-}count \leftarrow 0$
function BallStuck($w$, $v$)
    $a \leftarrow$ new action
    if $t < last\text{-}check\text{-}time + CheckInterval$ then
        return $Null$
    end if
    $last\text{-}check\text{-}time \leftarrow t$
    if Distance($ball$, $last\text{-}ball$) $< StuckThreshold$ then
        $stuck\text{-}count \leftarrow stuck\text{-}count + 1$
    else
        $stuck\text{-}count \leftarrow 0$
    end if
    $last\text{-}ball \leftarrow ball$
    if $stuck\text{-}count < StuckCount$ then
        return $Null$
    end if
    $t_0 \leftarrow t - CheckInterval \cdot (StuckCount - 1)$
    $a[cmd] \leftarrow$ STOP
    $a[reset] \leftarrow$ Pos($ball$)
    $v[loc] \leftarrow$ GAME_OFF
    $v[call] \leftarrow$ FORCE_START
    Output($\{\{t_0; t\}; \{\{\{\text{circle}; \text{Pos}(ball); StuckThreshold\}; ColorWhite; BallStuck; 0\}\}\}$)
    return $v$, $a$
end function
```

Algorithm 11: The algorithm that decides whether the ball has become stuck.

$$(\vec{p}_b^0 - \vec{p}_b^1) - (\vec{p}_b^1 - \vec{p}_b^2) = \vec{p}_b^0 + \vec{p}_b^2 - 2\vec{p}_b^1.$$

When this acceleration is above a certain threshold, the frame is treated as a potential collision: If any robot is within a certain distance of the ball, the closest robot is considered to have just touched the ball.

We implemented two additional algorithms for detection, Line and Backtrack, to improve the overall performance of touch detection. We found that all of these algorithms have fairly low false positive rates, so we combine them simply by reporting a touch when any individual algorithm does so.

Both of the new algorithms are based on, for each robot, the history of positions of the ball relative to that robot; that is, $\vec{p}_b - \vec{p}_r^i$ for each i for every frame. Let $\vec{d}_n$ represent this difference in the n^{th} world state before the current one (dropping the i index, since each robot is processed independently and identically).

The Line algorithm, which is visually demonstrated in Figure 6.3, checks whether the recent history of relative positions match a two-part broken line, with the intersection point near a robot at the appropriate time; the idea is that the trajectory of the ball consists mostly of straight segments, separated by times when it collides with a robot. Let R be the radius of the robot and r be the radius of the ball. Then, according to this algorithm, a robot has touched the ball when the following conditions are true:

- $\vec{d}_1$ is "between" $\vec{d}_0$ and $\vec{d}_2$, meaning $\left|\vec{d}_1 - \frac{\vec{d}_0+\vec{d}_2}{2}\right| < .1 \cdot |\vec{d}_0 - \vec{d}_2|$;
- $\vec{d}_4$ is between $\vec{d}_3$ and $\vec{d}_5$;
- the intersection of the lines $\overline{\vec{d}_0\vec{d}_2}$ and $\overline{\vec{d}_3\vec{d}_5}$ is $\mathit{MaxRobotRadius} + \mathit{BallRadius} + 30\,\text{mm}$ or less from the center of the robot; and
- $\frac{\vec{d}_2-\vec{d}_0}{|\vec{d}_2-\vec{d}_0|} \cdot \frac{\vec{d}_5-\vec{d}_3}{|\vec{d}_5-\vec{d}_3|} < .99.$

The conditions check that the last three observations form a straight line, the three before them form a different straight line, and the two straight lines intersect near the robot.

The Backtrack algorithm, which is visually demonstrated in Figure 6.4, checks whether tracing the current trajectories of the ball and each robot backwards in time results in a projected collision. The idea is that the changes from straight-line trajectories are usually caused by collisions with robots, so if the most recent trajectory looks like it came from inside a robot, it must be the result of a collision.

The algorithm starts by examining $\vec{d}_0$, $\vec{d}_1$, $\vec{d}_2$, and $\vec{d}_3$; if the line segment between any consecutive two of them contains a point with magnitude less than $\mathit{MaxRobotRadius}+\mathit{BallRadius}$, then it decides that the observations do not show a collision. (This check is an attempt to filter out false positives caused by chip kicks passing over robots.) Otherwise, the algorithm fits a constant-velocity trajectory to $\vec{d}_0$, $\vec{d}_1$, $\vec{d}_2$, and $\vec{d}_3$ by separately performing least-squares linear regression on both the x- and y-coordinates as functions of time and uses the results to extrapolate positions two frames further into the past; if either extrapolated vector has a magnitude less than $\mathit{MaxRobotRadius} + \mathit{BallRadius} - 10\,\text{mm}$, the robot is considered to have touched the ball.

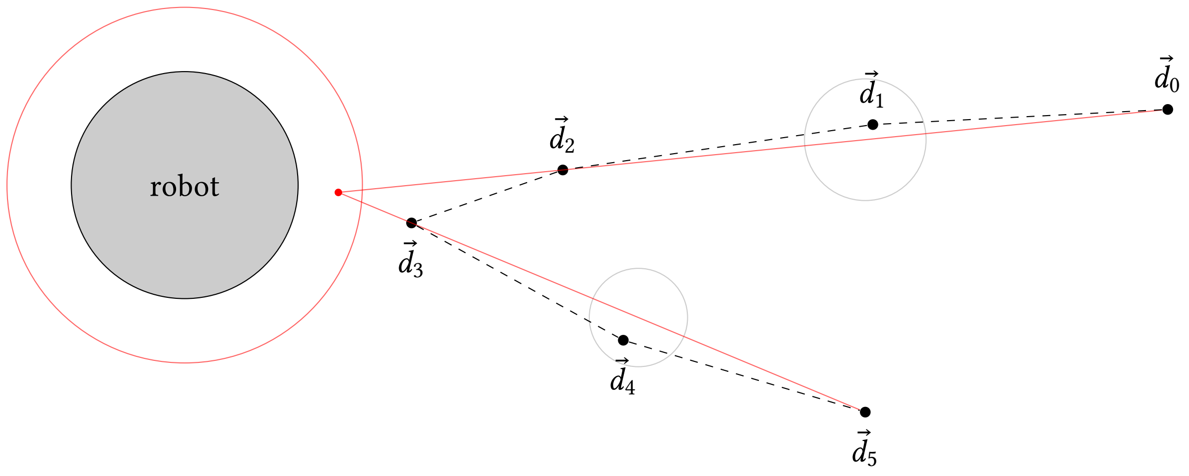


Figure 6.3: A visual description of the Line collision detection algorithm. The history is considered a collision if (a) $\vec{d}_1$ is inside the circle that is centered halfway between $\vec{d}_0$ and $\vec{d}_2$ and has radius equal to 10% of the distance between them, (b) $\vec{d}_4$ is inside the circle defined similarly by $\vec{d}_3$ and $\vec{d}_5$, (c) the intersection of the red lines is inside the red circle (which has radius *MaxRobotRadius* + *BallRadius* + 20 mm), and (d) the angle between the two red lines is sufficiently large.

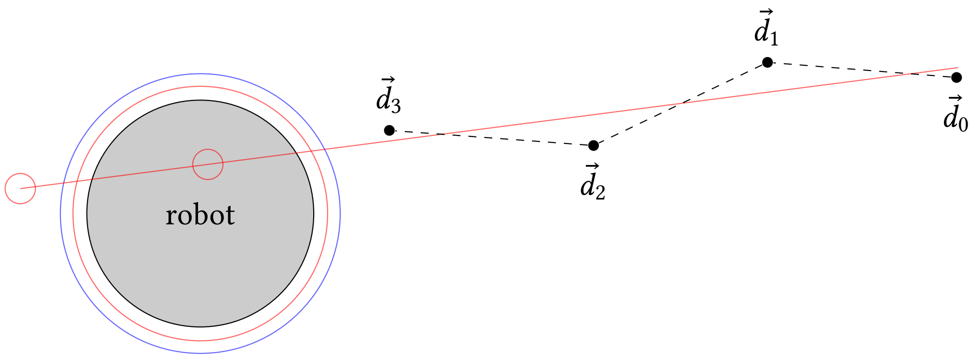


Figure 6.4: A visual description of the Backtrack algorithm. The red line indicates the result of performing linear regression on the x- and y-coordinates of the observed positions with respect to time and plotting the resulting functions against each other; the open circles represent the extrapolated positions at the times of the two frame times before the ones providing the observed positions. The history is considered a collision if (a) no segment between the observations has any point inside the blue circle (which has radius *MaxRobotRadius* + *BallRadius*) and (b) either extrapolated position is inside the red circle (which has radius *MaxRobotRadius* + *BallRadius* − 10 mm).

Robots settled

Another important event to detect involves deciding when the robots are ready to resume play by taking a kick. Waiting for the robots to settle down is not mentioned in the rules, and so there is no standard description of how to make the decision or, strictly speaking, any need to wait at all. However, this behavior is so common among human referees that we decided that it made sense to emulate it.

After examining the usual behavior of human referees, we decided to base this event on the speed of the robots. Each frame, the algorithm examines the speed of the robots and the ball. If all the speeds are below particular thresholds (possibly different for robots and the ball) continuously for a period of time, then the algorithm decides that the robots are ready. This algorithm is shown in Algorithm 12. The variable *start* represents the last time that the robots or ball were moving too fast. Line 6–Line 13 check whether all the speeds are below the appropriate thresholds; if not, they reset the *start* variable to the current time. Line 14–Line 16 check whether it has been sufficiently long since *start* was last reset, i.e., whether the speeds have been low enough for long enough. If execution continues past that point, then the robots are deemed to be ready; the remaining code determines how to respond.

The following code examines $v[call]$, which should have been used by a previous event to remember what call should be made next. If it indicates a free kick or the ready signal (for a kickoff or penalty), then a kick can now be taken, so the new automaton location should be SETUP. If $v[call]$ indicates a neutral restart, that means both teams are free to manipulate the ball and the location is GAME_ON. Finally, if $v[call]$ indicates a kickoff or a penalty, then there still needs to be a `READY` command sent, so the location is still GAME_OFF, but the next command to be sent is `READY`. Note that *start* is reset here, because we are waiting for a new transition period and settling. If it were not reset, then the event would fire again immediately on the next evaluation.

6.3.3 Challenges

The domain-specific logic is contained entirely within the detection of individual events, so it is critical that the event detection be reliable. In the RoboCup SSL, the primary obstacles are unreliable data and chip kick detection.

Although SSL-Vision returns precise and complete position data with high reliability, occlusion of the ball and the brittleness of the color calibration scheme mean that robots or the ball can be lost from vision for many frames at a time. Such failures are particularly a problem for judgments which involve contact between robots and the ball (mainly, touching the ball or dribbling it for too great a distance); vision of the ball is often completely lost during those situations, just when we need it the most.

The other main problem relates to chip kicks. Because SSL-Vision returns a ball position assuming that the ball is on the ground, the values it gives are not correct when the ball is in the air; they instead give the intersection of the ground plane with the line containing the camera and the ball. There are methods to detect when a chip kick is happening and determine the true trajectory of the ball, but we have yet to come up with an implementation that is sufficiently sensitive to be of use without giving too many false positives. As a result, many judgments based

Inputs:

- w: world state
- v: initial values of the autoref variables

Outputs:

- v: updated values of the autoref variables
- a: action to take if this event fires

Constants and helper functions:

- *BallSpeedThreshold*: the maximum allowable speed of the ball
- *RobotSpeedThreshold*: the maximum allowable speed of the robots
- Vel(x): the velocity of object x (a robot or the ball)

```
start ← −∞
function KickReadyEvent(w, v)
    if v[loc] ≠ GAME_OFF then
        return Null
    end if
    if |Vel(ball)| > BallSpeedThreshold then
        start ← t
    end if
    for r ∈ robots do
        if |Vel(r)| > RobotSpeedThreshold then
            start ← t
        end if
    end for
    if t < start + 1 s then
        return Null
    end if

    a ← new action
    a[cmd] ← v[call]
    if v[call] ∈ {INDIRECT, DIRECT, READY} then
        v[loc] ← SETUP
    else if v[call] = START then
        v[loc] ← GAME_ON
    else if v[call] = KICKOFF
     or v[call] = PENALTY then
        start ← t
        v[call] ← READY
    end if
    return v, a
end function
```

Algorithm 12: The algorithm to check if the robots are ready to take a kick to restart play.

on the position of the ball may be evaluated incorrectly in the presence of chip kicks, especially checking whether the ball has left the field: due to the typical positioning of the cameras above the center of each half of the field, most chip kicks near the edges of the field appear to take the ball outside the field at their peaks.

6.4 Results

In order to test the abilities of the autoref, we performed two types of evaluation. First, to test the overall stability and rule adherence of a game as judged by the autoref, we set the autoref up to run repeated simulated games, with two copies of the CMDragons team playing, and collected statistics about the results of those games. Second, to test the ability of the individual event detectors to operate on real data, we ran some of the detectors on data from games of RoboCup 2014 and compared the results to the calls made by the human referees at those games.

6.4.1 Running simulated games

Although not all of the rules of the SSL are currently implemented, the ones that are make up the great majority of rules that come into effect during games in practice. With the currently implemented events, it is possible to run an essentially full game in simulation, with kickoffs, appropriately awarded free kicks for ball exits and some fouls, and checks to ensure that the progress of the game is not delayed by a slow team or stuck ball. As a result, the autoref makes it feasible to run and gather information from a far greater number of rules-compliant games than would be feasible to monitor by hand.

As a simple demonstration of this capability, we ran many simulated games comparing two slightly different versions of our own CMDragons soccer team. In some tests, we enabled or disabled some new feature of the team, so that we could check whether having that feature gave the team an advantage. In others, one version of the team was handicapped, either by having fewer robots or by having the speed and acceleration of its robots restricted to a fraction of their normal values. Overall, we have run thousands of games without supervision.

We found that, when one team is simply handicapped, the other has a noticeable advantage; while this result is no surprise, the ability to verify this through experiment is made possible only by the presence of an autoref. The average scores as functions of the degree of handicap are shown in Figure 6.6; the error bars indicate the standard error of the mean after 100 games.

We also ran some tests to determine the effect of software features on performance. One of the features we tested was a zone-based attack strategy we developed and used in the RoboCup tournament last year; the other was a method for a robot to quickly turn while dribbling the ball, which we used for only part of the tournament because it was unreliable on real robots [Veloso et al., 2015]. The results are shown in Table 6.7; both features provided an advantage, as we hoped. However, this is one case where the limitations of the simulation environment must be kept in mind: the fast turning behavior worked very well in simulation, but we disabled it in reality because it did not work as well. The zone-based attack seems less likely to be dependent on the specifics of the simulation.

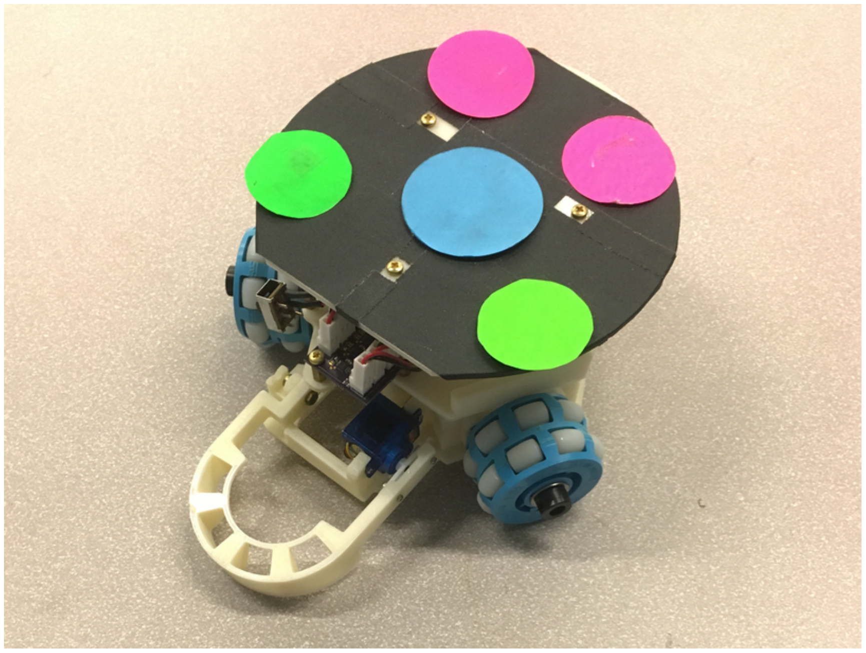
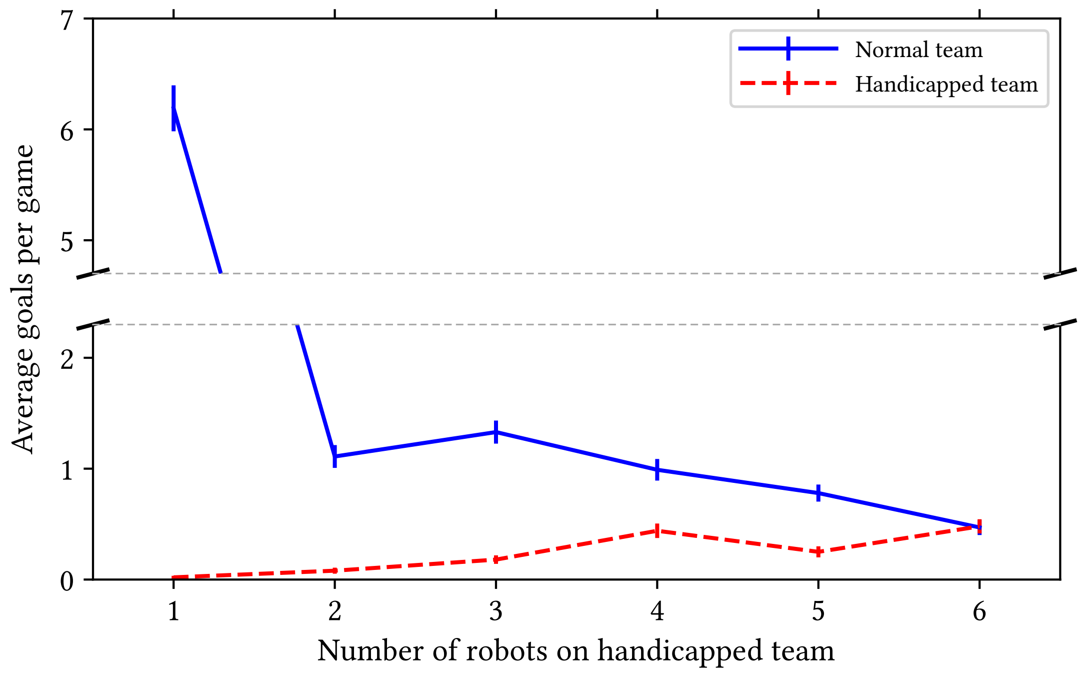


Figure 6.5: The specialized referee robot. The chassis is 3-D printed and the components were chosen with low cost in mind; the whole robot cost less than $200.

The modularity and common network interfaces of the various components of the SSL ecosystem also mean that the same autoref code is compatible with live games played on the physical robots. Of course, there are other concerns, such as robot batteries needing to be changed and the ball getting stuck outside the view of SSL-Vision. Even more important is the ability to position the ball for a kick. In simulation, we simply command the ball to be at a certain location, and it is there, but if we are to avoid requiring a human to position the ball every time there is a kick, we need some real-world analogy; using a ball-positioning robot is the natural choice.

We implemented the ability for CMDragons to listen to ball-positioning requests and use its own robots to fulfill them. When the ball needs to be placed, the goalie of one of the teams drives out to place the ball on its dribbler and pushes it to the desired location. With that procedure, the essential capabilities are already in place to run full games with little human intervention. We have also developed a specialized referee robot, shown in Figure 6.5[1], which we tested and displayed at RoboCup 2015. The main advantage of such a robot is that, for the sake of fairness, the rules prohibit player robots from surrounding the ball or removing all of its degrees of freedom, but such capabilities are of great use precisely for the task of quickly and repeatably positioning the ball on the field; our referee robot has an arm that can completely encircle the ball, holding it solidly in place in front of the robot. Ball placement since became a standard part of the league and was implemented by many other teams, as discussed in Section 6.4.3, so we ultimately discontinued development of the specialized robot.

[1]Thanks to Zongyi Yang for designing and fabricating the hardware.



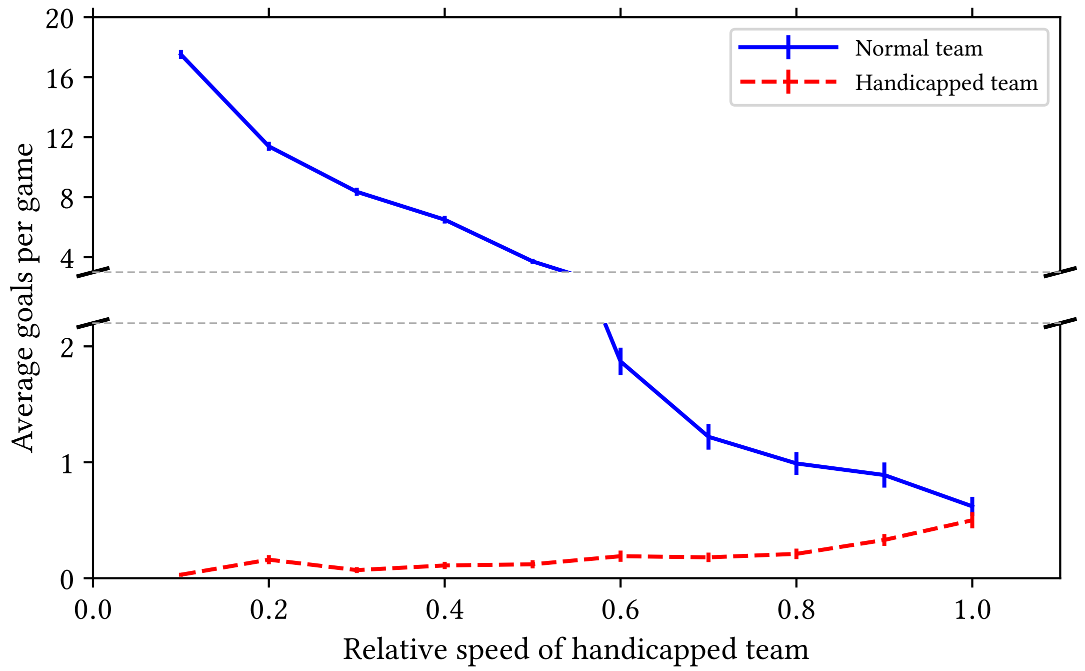


Figure 6.6: The results of comparing two versions of our team, one handicapped by having either fewer robots or reduced speed and acceleration. Each data point depicts the mean score and standard error of the mean of the two teams across 100 full games.

	Zone-based attack		Turn & dribble	
	Win %	Goals	Win %	Goals
feature on	38.00	0.58 ± 0.05	42.00	0.70 ± 0.05
feature off	12.67	0.18 ± 0.02	10.67	0.22 ± 0.03

Table 6.7: The results of comparing two versions of our team, differing by the presence or absence of a new feature. "Win %" indicates the percentage of games won by that team, and "Goals" indicates the mean number of goals per game, as well as the standard error of the mean. Each comparison is based on 300 games.

6.4.2 Comparison to human refereeing

To further test the accuracy of the autoref implementation, we compared its performance against the human referees of games from RoboCup 2014. For each of the quarterfinal (QF), semifinal (SF), and final (F) games, we manually reviewed our log files of those games[2], determined the times that a rule that the autoref can currently detect should have been applied, and recorded whether the referee actually made each call during the game. We then ran the autoref as if it were receiving the log data as real input, and compared the set of calls made by the autoref to the set of calls that were made.

We also noted which calls were made incorrectly by the human referee. In manually reviewing the logs, we have the advantage of being able to stop and examine the input from vision frame by frame, which we believe gives us enough certainty in most cases to tell whether a call was correct, even when our call differs from the referee's. Some cases remained ambiguous, usually when two or more robots obscured the ball while it exited the field; those cases were counted separately.

Ball exit calls

By far the most common call in a typical game is to stop the game when the ball exits the field and award a free kick to the opponents of the team which touched the ball most recently.

The results of both the manual annotation and the automatic detection by the autoref consist of a list of $\langle time, team \rangle$ pairs. A pair on one list is considered to match one on the other if their times are within 0.3 s of each other; each event can match at most one event on the other list. The results of this comparison are shown in Table 6.8. Annotated events which don't match to any detected events are described as *missed*, and *extra* events are the reverse.

There were no calls that were missed, and most of the extra calls made are due to chip kicks near the edge of the field; as discussed in Section 6.3.3, such kicks can cause the ball to appear as though it is outside the field when it is not. If the ball actually does leave the boundaries of the field during a chip kick, this could result in both a missed call and an extra call, as the call is made well in advance of the true exit, although that did not occur in any of these games. The only other extra calls were two during the final game that resulted from some brief span of

[2]We used the log files produced by our normal CMDragons code, which contain the object position information received by the autoref, as well as the commands given by the human referee and additional information about the team intelligence; we used only the vision information to review the games.

	QF	SF	F
last touch clear	47	59	52
last touch ambiguous	1	3	6
human referee			
correct	43	55	52
last touch incorrect	4	4	0
autoref			
correct	37	52	43
last touch incorrect	10	7	9
extra	0	0	2
chip extra	3	1	6

Table 6.8: Performance of the autoref for ball exit events as compared to human referees at RoboCup 2014. The *correct* lines refer to calls which were made at an appropriate time, with the subsequent kick awarded to the correct team; *last touch incorrect* refers to a call made at an appropriate time, with the kick going to the wrong team. The *extra* line for the autoref refers to calls made by the autoref that did not result from actual ball exits, except for those caused by chip kicks near the edge of the field, which are counted under *chip extra*.

spurious data from SSL-Vision, in which the ball rapidly flickered in and out of view and, for a few frames, appeared to be outside the field. (It may have gotten on top of a robot, which would both interfere with detection and, even when it was being detected, caused it to appear further out from the camera than it actually was, just as chip kicks do.)

Among the calls that whose presence was correctly identified, the proportion of team misidentifications by the autoref is larger than would be desired for the final version of the program, at about 15% of all ball exit calls. The touch detection algorithm is still incomplete and under development. Many of the incorrect identifications are also due to chip kicks; the ball may appear to pass very near or through a robot, registering as a touch, when in fact it passed over the robot.

We also compared the results of the different combinations of touch detection algorithms, as shown in Table 6.9. We show the performance of the previous baseline algorithm (in the first line) with the results of the human refereeing during the actual games (in the last line) and the various combinations of improvements (in between). The new algorithms we developed improved on what was previously available in our codebase, and it was promising to see that combining the different algorithms yielded the greatest improvement overall, though we have yet to equal the performance of a live human referee.

Maximum ball speed calls

The rule stipulating that the ball cannot be kicked above 8 m/s often goes uncalled when it is violated, since it is difficult for a human referee to accurately determine the speed of the ball. By contrast, the ball tracker used by the autoref already has accurate quantitative information about the ball, and so checking for violations of this rule is straightforward.

The autoref detected four occasions in the games where the maximum ball speed rule was

Algorithm	QF	SF	F
accel	.66	.66	.73
accel + line	.68	.73	.77
accel + backtrack	.79	.85	.83
accel + backtrack + line	.79	.88	.83
human	*.89*	*.93*	*1.00*

Table 6.9: The fraction of times that the autoref or human referee correctly identified the last team to touch the ball before it exited the field, among the times when it correctly called an out.

	QF	SF	F
true goals	3	2	2
false chip goals	1	3	1

Table 6.10: The counts of detected goals from the games. The only errors were false positives due to chip kicks.

violated, none of which resulted in corresponding calls by the human referee. In the quarterfinal game, our opponents violated the rule three times in the first half, kicking the ball in excess of 9 m/s, well above the stated limit; in the second half, we ourselves kicked a ball slightly too fast. There was also one spurious identification in each half, caused by false vision reports of a ball when a human stepped onto the field. (In practice, these would not be likely to cause any issues, since there will be humans only on the field while the game is stopped. Of course, it is also possible to improve the detector to filter out such detections to begin with.)

Goals scored

Finally, we tested the detection of goals scored. Chip kicks are a major confounding factor here as well; many kickoffs are taken by simply chip kicking the ball directly over the opponent goal, which looks rather like the ball going into the goal.

The detector correctly identified all of the goals that actually occurred in the games we examined, and there were no false positives apart from the ones caused by chip kicks.

6.4.3 Anecdotal evidence of reactions from RoboCup 2017

Finally, we conclude with some anecdotal evidence: human reactions to the operation of the autoref at RoboCup 2017. At that tournament, the TC of the SSL (which included the author of this thesis) invested effort into having the autorefs (the one described here, as well as the one by ER-Force) actually running and presenting output to humans during the games.

As part of the procedure for using the autorefs, we created an algorithm for finding a consensus between the two autorefs: only when both of them made the same call within a short time interval did the consensus algorithm actually display the call and (when applicable) forward it to the teams. During observation of the performance of the autorefs and consensus algorithm over a number of games at the tournament, we observed multiple cases where one autoref but not the other produced a false positive, causing the consensus algorithm to correctly reject the detection. In general, we believe it will continue to be useful to allow for the possibility of maintaining and combining multiple independently-developed autorefs to improve robustness against incorrect detections.

At an open meeting of team members during the tournament, reactions to the autoref were generally positive; those present agreed that the presence of the autoref during games helped make calls when the human referees were unsure. The overall feeling was that though the autorefs might not be truly competitive with human referees yet, the humans were ready to accept an increasing presence of the autorefs moving forward.

Finally, we would like to specifically mention the reaction of observers to the combination of the autoref with automatic placement of the ball by the competing teams. “Auto-placement”, as it is usually called, involves a special referee command that an autoref can send to teams, which comes with an associated point on the field; a team should respond by commanding its robots to place the ball at the specified point. Auto-placement began as a RoboCup technical challenge in RoboCup 2016, and some teams maintained the ability to perform it at the 2017 competition.

During some games, we enabled auto-placement when at least one of the teams could respond, either at the request of the teams or of the TC. Combining the autorefs with auto-placement is the closest the SSL has come to fully-automated games. Despite the rapid pace and complexity of normal gameplay, successful auto-placement garnered an immediate reaction from the spectators, with the crowd often cheering and applauding after a correct placement. Apart from the effect of sheer novelty, we believe the spectators were reacting specifically to the visible movement toward full automation of the game, helping to demonstrate the worthiness of further development of autorefs for the SSL.

6.5 Visualizations

As with the other robot domains, we finish our discussion of the autoref by returning to the theme of visualization and seeing how it can apply to the autoref; we show several examples of visualizations that we can produce from the autoref’s event detectors. We still can, even with the past-oriented nature of the autoref, work with recorded videos to take advantage of the ability to display future information on a video: we can connect the future and the present within the video, just as we do when showing plans for future actions with the other robots. We describe the drawings below in this context, since we consider it to be the best use of such visualizations for the autoref, but it is also possible to display drawings as they come on top of a live video; in that case, we can show the drawings corresponding to an event after it has been detected. In Section 6.5.1, we provide a further discussion of the internals of the autoref as they relate to the visualizations we discuss, and then describe the specific resulting visualizations in the following sections.

Since the autoref works in the same domain as the soccer visualizations described before, we can imagine including the autoref and soccer visualizations on the same video. We are inspired by the idea of “instant replays” often used in human sports, where a video replay of an event may be used to show viewers what happened to cause the referee of the game to make a call, or perhaps even to help the referees make a difficult call in the first place. In this case, we envision the following mode of operation for replaying a video of a soccer game: we show the game with the visualizations from a soccer team at first. When the autoref makes a call, someone viewing the video may wish to question why it did so. We then change to a separate mode for the viewer, where it jumps to an earlier time and switches from showing the visualizations of soccer to showing those of the autoref. Once it reaches the time it had originally been showing before the jump, it returns to showing the normal soccer visualizations.

6.5.1 Autoref implementation internals

The hybrid automaton structure described in Section 6.2 and the definitions of the events as functions operating on sets of variables match closely to our actual C++ implementation of the autoref. For each event, there is an *event detector* object (or just *event object*): an instance of a class that encapsulates the logic to detect whether the event has occurred, given successive world states. A global autoref object manages the individual event detector objects, supplying them with successive world states and handling the results of event detections; the autoref object has member variables that describe the state of the autoref (i.e., its current automaton location and the other auxiliary automaton variables, as well as generally useful information, such as the last robot that touched the ball), corresponding to the set $\mathcal{V}$; the event objects may also contain member variables relating to their specific events; these variables correspond to the summaries $\mathcal{S}_e$.

Each event object has a `process` method, which takes in a new world state as its argument. The method performs computation using the world state and the available variables, possibly updating the event object's member variables. It returns an object that describes how the autoref should respond. This return value may contain updated values for the autoref variables (also corresponding to $\mathcal{V}$) and possibly a new command to transmit to the teams and position to place the ball (corresponding to $\mathcal{A}$).

For the purposes of visualization, we can take the values of the variables used by the event objects and map them into drawings, much like we did with the other robot domains. Many of the events store and deal with locations, lending themselves to our position visualization.

6.5.2 Ball touched and ball out

For a ball touch event, we draw a circle around the location of the ball at around the time of the touch. When the autoref detects a ball touch, it creates a persistent frame whose start timestamp is equal to the time of the detected touch and whose end timestamp is a small time (0.5 s) after.

We also combine this drawing with a ball out event: when the autoref detects that the ball has exited the field, it creates a persistent frame containing a line segment whose endpoints are (a) the point where the ball was touched and (b) the point where it exited the field. This additional drawing serves the aim we have described of bringing future information to earlier frames in a video: one of the most frequent decisions in a game is determining which team was the last to touch the ball when it exits the field. Figure 6.7 and Figure 6.8 show examples of these visualizations.

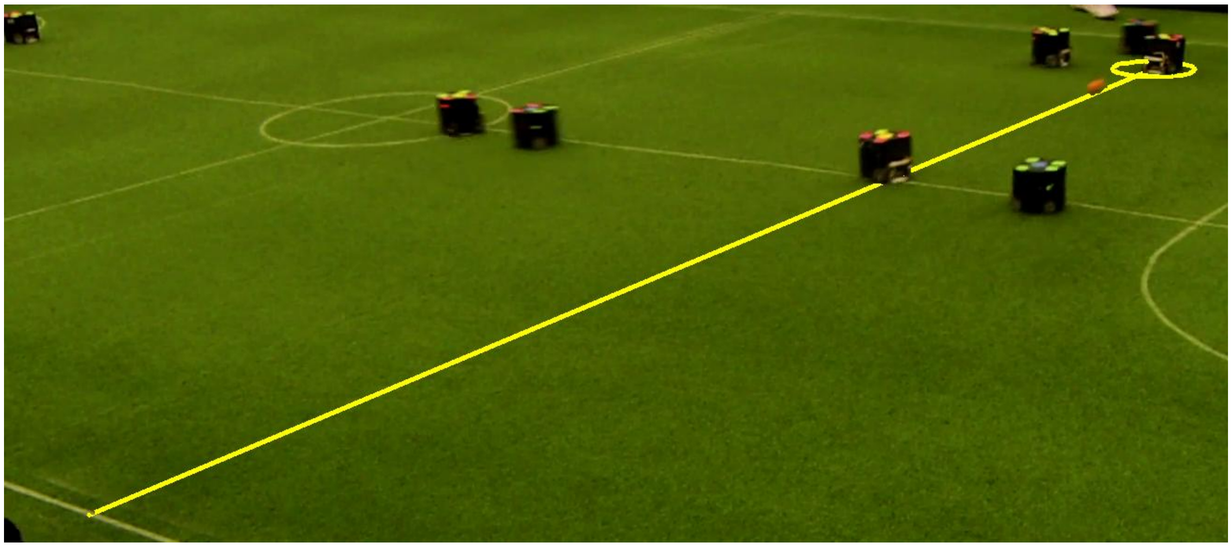


Figure 6.7: An example of the visualization resulting from the ball touching a robot and exiting the field. While viewing the video, we can see at this point that the ball will exit the field without touching any more robots after touching the robot in the circle, even though another robot is along the current path of the ball.

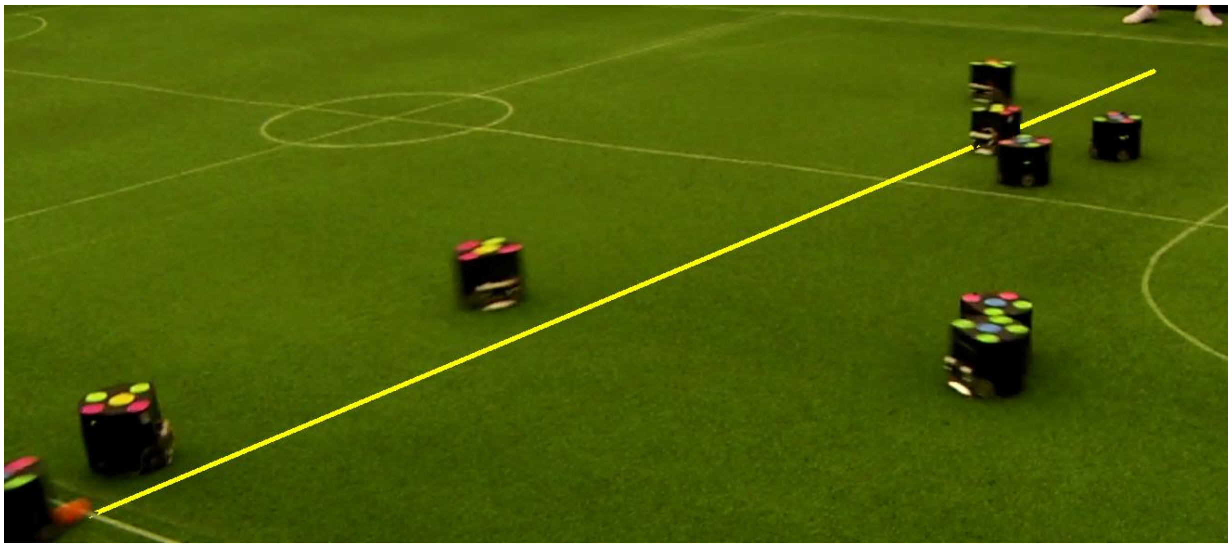


Figure 6.8: The moment when the ball exited the field after the time shown in Figure 6.7, showing that the ball indeed continued to the exit point indicated by the line.

6.5.3 Ball kicked too fast

The detection algorithm for whether the ball was kicked at too high a speed checks, each frame, whether the distance from the ball's current location to its position in the last frame is greater than the time between frames multiplied by the maximum allowed speed. If it is so for four consecutive frames, then the algorithm declares that the ball is moving too fast and that the last robot to touch the ball committed a foul.

The variables used by this event are the current ball position, the last ball position, and the count of frames for which the speed has been too high. For the purposes of visualization, we also store the previous three ball locations, for a total of five positions, which are all that influence the detection of the event. When the event fires, we draw circles around all of the saved locations, showing the distance between them. Figure 6.9 shows this visualization.

6.5.4 Ball stuck

The ball stuck event, as described in Algorithm 11, fires when progress of the ball has been stopped for a long-enough period of time.

For this event, the autoref creates a persistent frame containing a circle at the location that the ball has when the event fires; the, but starting at the beginning of the interval that defines the event. In watching a video playback after the fact, the appearance of the circle at a moment in time lets the viewer know about the future of the video.

The detection of the ball stuck event relies on a relatively long history, compared to other events, most of which rely on information from a short period of time. By creating a drawing that exists from the beginning of the relevant part of the history, when a realtime observer would still not be able to know for sure whether the event would happen, we can visibly enhance the ability of the video to provide information about the computations of the autoref. Figure 6.10 shows an example of this visualization.

6.5.5 Distance from ball during stop

Another of the rules that is hard for humans to enforce precisely is that when the game is stopped, robots must remain at least 500 mm away from the ball. Human referees cannot precisely determine the distance from the robot to a ball by looking at the field, but it is easy for a system with access to SSL-Vision. In the CMDragons viewer, we display a gray circle of radius 500 mm around the ball any time the game is stopped, providing a concrete reference for human viewers to look at. We can use the autoref visualizations to provide a similar display when the game is stopped; when the distance is actually violated by a robot, it changes to red to show the violation of the rule. Figure 6.11 shows an example of the visualization resulting from a violation.

6.6 Discussion and summary

In this chapter, we presented our autonomous referee (autoref) for the previously-introduced RoboCup SSL. We described the formalization of the game as a hybrid automaton; any referee,

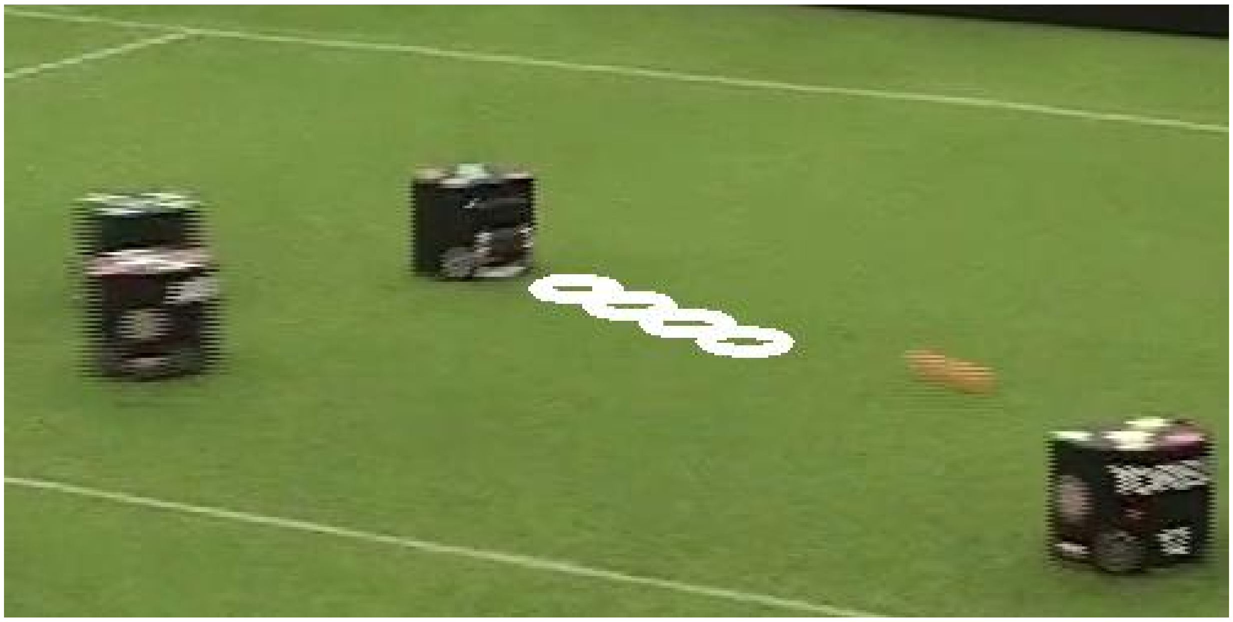


Figure 6.9: An example of the drawings that could result from the ball being kicked too fast, showing a sequence of observed ball positions in consecutive frames that are too far apart for a legal kick.

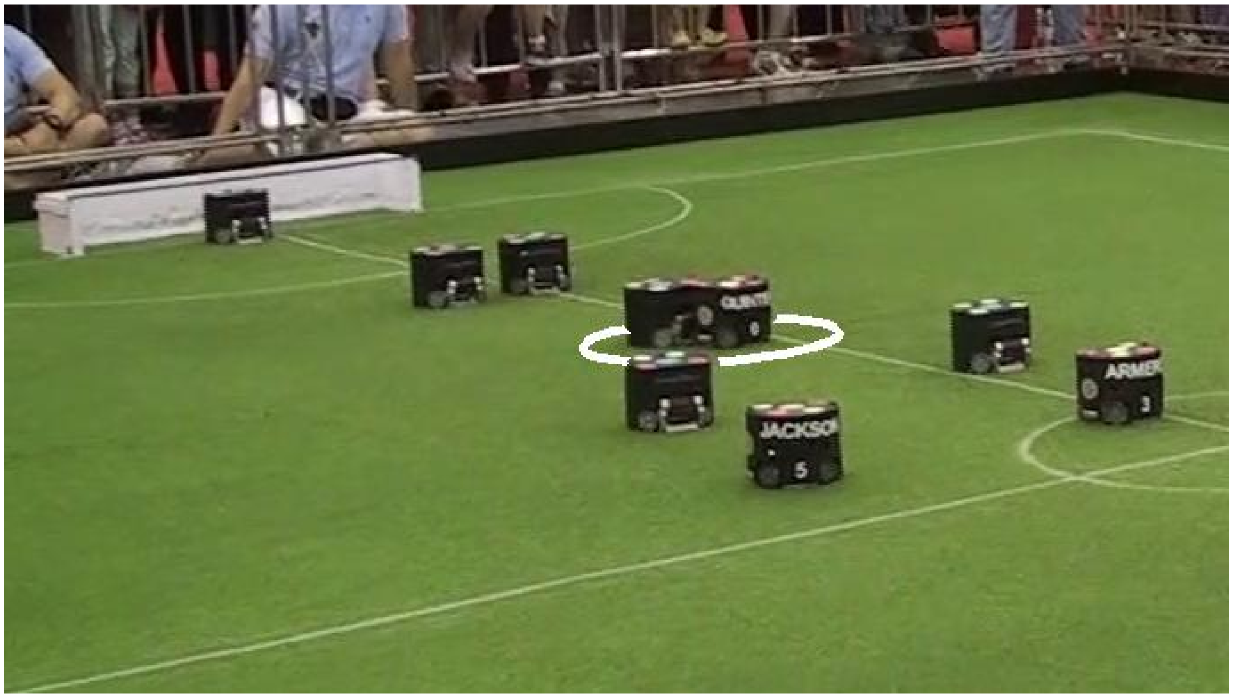


Figure 6.10: An example of the visualization shown when the ball is in the process of becoming stuck during play. The ball is stuck between the two robots in the white circle.

whether human or automated, must essentially model the execution of this automaton according to its information about changes in the state of the world. We then described in detail how we implemented an autoref according to the structure of the game automaton, by converting the transitions of the automaton into discrete events that we implement in code. We gave results of using the autoref with full SSL games, including the results of running repeated games in simulation, the efficacy of the event detections on real-life games, and the human reactions to use of the autoref in real games. Finally, we turned back to the visualization structures of Chapter 3 to the autoref; although the autoref is less planning-oriented than the other domains, we found fruitful places to apply the visualizations with it.

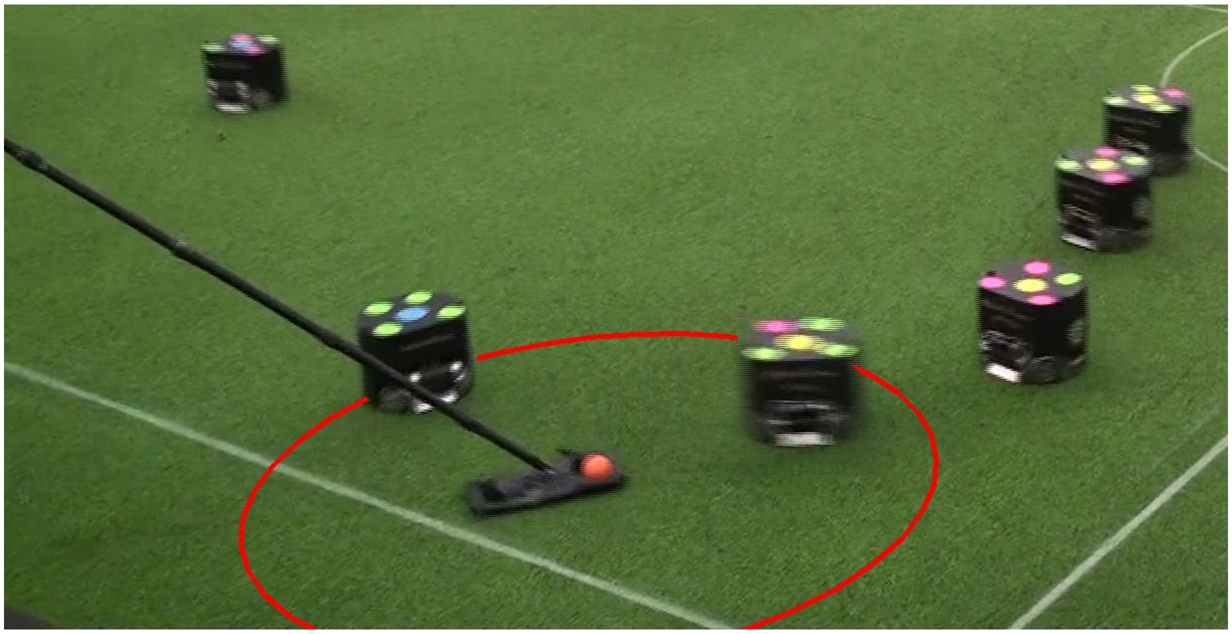


Figure 6.11: An example of the visualization shown when a robot is too close to the ball while the game is stopped; the red circle shows the minimum allowed distance from the ball when the game is stopped.

Chapter 7

Related work

This chapter presents an overview of previous work related to the topics of this thesis. In Section 7.1, we discuss work related to our development of an automatic referee for SSL games. In Section 7.2 and Section 7.3, we discuss the fields of visualization and mixed reality.

7.1 RoboCup SSL game analysis and refereeing

In addition to the development of the competitive RoboCup teams, there has also been work on automatic analysis, both online and offline, of games played in different RoboCup leagues. Examples of the goals of such work include classifying the roles of the robots in the game, following the ball, providing commentary on the progress of the game, and making decisions related to refereeing the game.

Vail et al. [2007] investigated using conditional random fields for activity recognition of different robots based on a stream of their positions over time, much like the data available in the SSL. For automatic refereeing, it is also necessary to perform recognition of a sort: a referee's fundamental task is to interpret the situations described in the rules and determine when they are happening. However, most of the relevant situations for refereeing are amenable to simple handmade detectors. More advanced techniques for classification may be useful for more difficult situations, such as those that are based on attributing intent or negligence to the teams.

Many other authors have addressed the problem of online processing of RoboCup games, in pursuit of various aims. A simple example of such an aim is the automated cameraman for the RoboCup Middle Size League (MSL) described by RFC Stuttgart [Käppeler et al., 2010]. The cameraman shares information with a team to keep a camera pointed toward an object of interest at all times during the game. The object of interest is usually the ball, but may instead be a goal or goalie when a robot makes a shot on goal.

A more involved task is producing engaging, human-like commentary of games in real time. Rocco [Voelz et al., 1999] and Mike [Tanaka-Ishii et al., 1998], commentators for simulation games, use hierarchical declarative event systems to produce discrete events which may be the topic of utterances. CMCast [Veloso et al., 2008] is a commentary system for soccer games played by Sony AIBO robots; it is embodied in two humanoid robots which can autonomously move around the field, localize themselves, track the ball, detect game events, and give commentary appropriately. All of these commentators revolve around a concept of discrete "events" derived from the continuous state of the game, similar to the event detectors we describe later. Commentating requires more understanding of the specific game under observation than does acting as a cameraman, but less than full refereeing; it shares the problems of event detection with refereeing, but the focus of such systems is typically more on utterance selection and audience interaction.

There have also been other efforts to provide automated refereeing for RoboCup games. For many years, the simulation leagues have had official autorefs integrated with their simulation servers, which have grown to be encompass many rules at this point, although games still require human referees for the more subjective events. In both the 2D [Chen et al., 2003] and 3D [Simspark wiki: Soccer simulation] simulation leagues, the autorefs can handle

- managing the length of game halves,
- resetting robots to legal locations when the game is stopped,
- detecting when a free kick is taken,
- awarding the proper kicks when the ball leaves the field or enters a goal, and
- detecting offsides situations.

Additionally, the 2D referee can detect backpasses, but relies on a human referee to detect players surrounding the ball or a team blocking its own goal with too many players. The 3D referee can detect when the players have formed a cluster around the ball that should be broken up and when a team's goal is being blocked by too many of that team's robots; in either case, it teleports the offending robots outside the field.

The autoref we describe, when running with a simulation of real robots, is similar to these referees in scope. However, referees that are designed purely for simulation and execute within the simulator itself have the advantage of being able to rely upon having full, direct access to the state of the world, in terms of both reading (i.e., determining the current positions of world objects) and writing (i.e., instantaneously resetting the robots or ball to desired positions). As a result, detection of many events is much less error-prone for such referees.

For real robots, Arenas et al. [2009] developed a refereeing robot that watched games of the RoboCup 2-Legged Standard Platform League and the Humanoid League. By necessity, that work mainly focused on the lower-level aspects of event detection from vision, since those leagues do not have the standard centralized vision system of the SSL, but the ultimate aim is equivalent to ours for those leagues. For the MSL, Tech United Eindhoven [Schoenmakers et al., 2013] briefly described an autoref that can detect when the ball leaves the field, automatically signal the robots to stop play, and wait for a human to position the ball in the correct location. Their algorithm is like a simplified version of the one described here. The RoboCup Logistics League (RCLL) has also recently gained an autoref [Niemüller et al., 2013], which was initially developed at the same time as the work described here. The kinds of events that need to be detected for that league are rather different, but the ultimate goals described there are much the same: taking a difficult job off human referees, ensuring objectivity in judging, and benchmarking the systems used in the game. The RCLL autoref communicates with team robots and neutral machines over the network, awarding points when robots complete particular tasks using the game pieces, which abstractly represent materials in a factory. The communication the autoref receives consists of discrete messages, primarily about completed tasks, so that it does not need to perform its own processing of real-world data to extract relevant events, as ours does. However, the rule-based system at the core of the RCLL autoref achieves a similar purpose to the structures described here, and could be used as the basis of as alternate implementation.

Finally, at least two existing projects have worked toward automatic refereeing for the SSL specifically. At RoboCup 2014, the SSL released the technical challenge of detecting certain relevant events from the game, such as robots colliding with excessive force or kicking the ball too fast [RoboCup 2014 Technical Challenges]. ER-Force, the SSL team of University of Erlangen-Nuremberg, has published the code it developed for the challenge [Robotics Erlangen autoref]. Also, the open-source project `ssl-autonomous-refbox` [ssl-autonomous-refbox] uses Prolog to declaratively define the rules as predicates over game states. The ER-Force code can only handle detecting those isolated events, while `ssl-autonomous-refbox` is limited to observing a game and lacks the ability to handle the transitions related to restarting the game after a stop; our autoref provides a more complete solution than either of them by being able to handle a full game from start to stop, transmitting the appropriate commands to teams.

7.2 Visualization

Visualization, in the most general sense, refers to any means of using graphics or images to convey information. Visualizations have been made on an ad hoc basis for a long time, but organized study of the principles behind it has become much more common in recent times. A seminal modern figure in visualization is Edward Tufte; his book *The Visual Display of Quantitative Information* [Tufte, 2001] provides a history of visualization, as well as extensive guidelines on designing useful and comprehensible illustrations of all kinds of data. Tufte codified and demonstrated general principles that are widely applicable to both print- and computer-based visualizations, such as removing extraneous material on the page and ensuring that the variation in a visual display has only as many dimensions as the data do.

There are several categories of science-related visualization; the most relevant to our work are scientific visualization and information visualization. Information visualization involves displaying abstract data that do not have an inherent spatial representation that can be mapped directly onto a display, while scientific visualization involves displaying data that do have some inherent spatial relations [Friendly and Denis, 2001]; the data are often measurements of some physical process.

Much work has also focused on visualization as applied to computer programs. This work is generally referred to as *software visualization*, of which *algorithm animation* is a part. Software visualization refers to any method of visualizing aspects of computer programs, such as the relationships between components, the text of the code itself, or a program's behavior over time [Ball and Eick, 1996, Price et al., 1992].

Animation has long been an invaluable tool for understanding algorithms. In algorithm animation, the changes over time in the state of a program are transformed into a sequence of explanatory graphics. Done well, such a presentation can explain the behavior of an algorithm much more quickly than words. BALSA [Brown and Sedgewick, 1984] is an influential early framework for algorithm animation. It introduced the idea of *interesting events*, which arise from the observation that not every single low-level operation in an algorithm should necessarily be visualized. With BALSA, an algorithm designer inserts calls to special subroutines inside the algorithm; when execution of the algorithm reaches the calls, an interesting event is logged, along with the parameters to the call. The designer then writes a separate renderer that processes the log of interesting events into an actual animation to display.

More recently, there have been several websites that collect examples of algorithm animation and provide tools for creating new animations [Algomation, Algoviz, Visualgo]. They follow BALSA's paradigm of interesting events and provide easy-to-use environments for implementing and displaying algorithms.

We are performing a form of algorithm animation here, but with a slightly different focus: most animations are designed with education in mind and delve into the details of an abstract algorithm, rather than executing on a physical robot. Additionally, the typical lifecycle of an algorithm on a mobile autonomous robot is different from the standalone algorithms that are usually animated. In most animations, a single run of the algorithm from start to finish results in an extended animation, and each interesting event corresponds to some interval of time within it. For an autonomous robot, algorithms are instead often run multiple times per second, with each run contributing a tiny amount to the behavior of the robot, and it makes sense for each frame of an animation to depict all the events from one full run. An example of this kind of animation comes from the visual and text logging systems employed by teams in the RoboCup Small Size League, a robot soccer league [Kitano et al., 1997]. The algorithms behind the teams in the league consist of many cooperating components, each with its own attendant state and computations running 60 times per second. As a result, understanding the reasons behind any action that the team takes can be challenging, necessitating the development of powerful debugging tools in order for a team to be effective.

For text-based log data, Riley et al. [2001] developed the idea of "layered disclosure," which involves structuring output from an autonomous control algorithm into conceptual layers, which correspond to high- and low-level details of the agent's state. Teams have also developed graphical logging in tandem with the text output: the algorithms can output lists of objects to draw, and we can view both this drawing output and text output either in real time or in a replay fashion. The existence of these tools speaks to the need for informative debugging and visualization when dealing with autonomous agents.

7.3 Mixed reality

Mixed reality is a general term originally defined by Milgram and Kishino [1994] to encompass any system that "involves the merging of real and virtual worlds". The term was introduced to extend and generalize the idea of *augmented reality*, which was rapidly coming into use to describe new systems at the time. Milgram defined a reality-virtuality continuum; fully virtual and fully real environments exist at opposite ends of the spectrum, with combinations in the middle. Augmented reality systems are those near the reality end; they are typically displays, often but not necessarily head-mounted [Milgram and Colquhoun, 1999], that show the real world itself (either directly or through a video) along with additional information on top.

7.3.1 Augmented reality

Augmented reality has seen a wide variety of uses, both in relation to robotics and otherwise; it allows a user's view of reality to be enhanced without being entirely replaced. Azuma [1997] listed a variety of uses of augmented reality, including those for medicine, manufacturing, robotics, entertainment, and aircraft control. One common use of augmented reality within robotics is to provide enhanced interfaces for teleoperated robots, such as interfaces for controlling a robotic arm that overlay 3-D graphics onto a video view of the arm [Kim, 1996, Milgram et al., 1995]. Such interfaces can a view of the predicted trajectory of the arm after a sequence of not-yet-executed commands, allowing for much easier control, e.g., in the presence of signal delay between the operator and the arm.

Our past work shares the 3-D registration and drawing that are common to augmented reality systems, although most definitions of augmented reality require an interactive or at least live view, which we do not currently provide. Amstutz and Fagg [2002] developed a protocol for communicating robot state in real time, along with a wearable augmented reality system that made use of it. They demonstrated an example application that overlaid information about nearby objects onto the user's view, one that showed the progress of a mobile robot navigating a maze, and one that showed the state of a robotic torso. Though they focused mainly on the protocol aspect, their work has similar motivations and uses to ours.

An instance of a popular early commercial product related to augmented reality appears in television broadcasts of American football games [1st & Ten Graphics, Honey et al., Rosser et al., 2000]. Many such broadcasts add a virtual down line or other information for the benefit of viewers; the displays are designed to be located and occluded as if they were present on the playing field. Such displays are mature and capable of high precision in positioning and occlusion, but they require pan and tilt sensors to be attached to all cameras and assistance from a crew of several humans in order to keep track of changes in lighting conditions.

There has also recently been a surge in the development of augmented reality commercial devices and platforms, such as the Oculus Rift, Apple ARKit, Microsoft HoloLens, Samsung Gear, and Google Tango, with a corresponding wave of creation of games and other applications for them.

Other recent academic work has involved using physical projections to aid in the understanding of the behavior of autonomous robots. For instance, Chadalavada et al. [2015] demonstrated a robot that signaled its intentions by projecting its planned future path on the ground in front of it. They found that humans in the robot's vicinity were able to plan smoother trajectories with the extra information, and gave the robot much higher ratings in attributes such as predictability and reliability, verifying that describing an agent's internal state is valuable for interacting with humans.

One prior publication deserves particular note here: the doctoral thesis of Collett [2007] described an augmented reality visualization system which shares many characteristics with we focus on combining the drawing with the ability to display and animate details such as the future plan of the robot, rather than only the low-level sensor and state information from the robot.

7.3.2 Augmented virtuality

Toward the other end of the reality-virtuality continuum is *augmented virtuality*, a term also introduced by Milgram and Kishino [1994], in which the primary environment is virtual; we also discuss here systems in which the real and virtual components are on roughly equal footing, which are often simply referred to as “mixed reality” in the literature.

Robert and Breazeal [2012] and Robert et al. [2011] demonstrated two systems based on mixed reality: an educational robot that appears to children to disappear and reappear inside a screen, and a game where players control a physical robot that bounces a projected ball to a wall, where it appears within a projected world.

The concept of hardware-in-the-loop testing (HIL) shares clear similarities with augmented virtuality, though its origins are different. Hardware-in-the-loop testing involves emulating a complex mechanical system by constructing part of it and simulating the rest, with appropriate actuators and sensors to interface between the two portions [Fathy et al., 2006]. With an appropriate design, HIL greatly reduces the expense of evaluating a new device by reducing the amount of hardware that must be built, shortening the time needed for development cycles, and enabling the reproducible generation of extreme situations that would be difficult to produce for an entire system. HIL is widely used in many disciplines, including automotive engineering and space robotics [Fathy et al., 2006, Isermann et al., 1999, Takahashi et al., 2008].

Chapter 8

Conclusion and future work

With this chapter, we conclude the thesis by providing a summary of the contributions it describes and a discussion of possible directions in which future research based on the work could extend.

8.1 Contributions

The key contributions of this thesis are as follows.

Autoref We developed and implemented an autonomous referee for the RoboCup SSL and tested it extensively with repeated simulated games, online operation with real games, and offline processing of previous games. The autoref centers around a model of the game as a hybrid automaton; by processing input from the world, it estimates what transitions should occur within the automaton, and computes the game commands that it should transmit accordingly. We implement the evolution of the automaton by breaking down its possible transitions into a set of discrete events, each of which is itself implemented as a discrete piece of code within the autoref.

SSL defense We developed and implemented an algorithm for the RoboCup SSL that successfully provided the defensive capabilities for our SSL team, CMDragons. The defense revolves around the concept of threats, which are distinct possible avenues for the opponent team to score a goal on our team. We used successive versions of this defense at the RoboCup tournament from 2013 to 2016, providing strenuous real-world testing of its performance; in 2016, scored 48 goals across all of our games while conceding none, providing a powerful real-world example of the strength of the defense.

Domain-independent visualization language The central aim of this thesis is to enable the automatic creation of augmented reality visualizations that describe and elucidate the reasoning of autonomous robots. To achieve that goal, we started with a description of a domain-independent format for describing any set of visualizations with timing information. We then successfully showed how to apply that format to diverse autonomous robots.

Concrete visualization software To demonstrate the practicality of the visualization concepts we outlined, we also implemented them as concrete software, including both the creation of visualization logs by different autonomous robots and the reading and rendering of those logs by later software.

8.2 Future work

As the main thrust of this thesis, we introduced a means of combining visualizations with videos that we applied to four different robotic algorithms. Though the concepts and implementations we produced form a complete system that has already demonstrated usable visualizations, there is still considerable room to further develop them.

8.2.1 Visualization

Automatic extraction of drawings from existing algorithms As of now, we assume that any algorithm under examination already includes specifications of how to create relevant visualizations from the quantities manipulated in the algorithm. However, creating these visualizations requires extra thought by the creators of the robot algorithm, and we would like to remove this extra step. It should be possible to apply analysis to the code of any robot algorithm to automatically extract a set of potentially-interesting quantities from the algorithm (e.g., named variables in the program) and annotate the source code of the program to create drawings depicting those quantities.

As an extension, we could apply more advanced analysis to the program to create additional links between the quantities describing their relations to each other in the structure of the program. In the current version, choosing entities and, especially, filter levels seems to us to be the most tedious part of the addition of visualizations. Automating the generation of the drawings from the algorithm itself would reduce the barrier to using our visualization systems.

Indirection for display-time adjustments The current description of the drawing storage format includes fully concrete descriptions of the drawings, with the exact shape and color of each drawing. Ideally, we would shift to a more semantic representation with additional indirection: the stored format could represent reasoning quantities in a higher-level form, and the final transformation into drawing primitives could be specified when the log is displayed, rather than when it is created. The flexibility of display we have described is a big part of the motivation for the work, and adding this indirection would further improve that flexibility.

User studies Throughout this thesis, we have laid the groundwork for creating the visualizations we describe, with an eye toward the idea that the output should be useful for humans to view and help them understand the operation of the robot algorithms in use. We focused on providing these capabilities and on the specific technical possibilities of how we could create a general visualization system and apply it to the robots, but considered it out of scope to go on to evaluating our visualizations with human users. Eventually, we should move on to doing such evaluation to further refine our creation of visualizations, in terms of both the general frameworks we use and the drawings chosen for any particular robot. Anecdotally, we have already found that viewing the visualizations certainly can make the operation of the robots clearer.

Three-dimensional settings We have restricted ourselves to two-dimensional drawings and robots that function within an essentially two-dimensional space; of course, many robots do not fit within those constraints.

To apply to a three-dimensional setting, most of the drawing aspects described here would need to be overhauled: a homography no longer suffices to describe coordinates in three dimensions, and more correspondences would be necessary to establish the coordinate mapping. The issue of masking also becomes more complex, since a given pixel is no longer either foreground or background; it may obscure some drawings but not others, depending on their positions in space; for fully-correct drawing, we would need a depth map corresponding to the camera image. The problems of determining the necessary information for transformation and masking have long been studied in many contexts, both academic and commercial, assuming the presence of a wide variety of equipment; we considered such study out of scope. On the other hand, the ideas of categorizing and filtering different drawings, a more fundamental contribution of this thesis, continue to apply without modification to this setting, since they are independent of the physical space used by the robot and the drawing.

Drawing extensions There are also natural extensions to the drawing language to allow for more varied drawings. Most such extensions do not fundamentally change the nature of the drawing process, but would make the language more convenient and flexible to use; examples include the addition of more types of geometric primitives (e.g., ellipses) or drawing parameters besides color (e.g., line thickness or fill color). A more substantial addition would be the creation of a new type of drawing primitive containing text and a position instead of a geometric shape. The drawing process for text could be even simpler than the one for shapes, consisting only of computing the position corresponding to the position, shifting it upward in image space, and drawing the text in image space at the shifted position.

8.2.2 Autoref

Smoother interaction with humans The autoref as we have described it is a self-contained model of the game that assumes that no other agents are making calls or trying to decide the non-physical game state; i.e., it assumes that no human referee also wants to make calls or dispute the calls of the autoref after the fact. For the RoboCup 2017 tournament, the autorefs ran at games, but in a limited mode that allowed them only to signal events during game on, which sidesteps this issue. Moreover, due to the absence of any way to give feedback to the human referees[1], the referees usually chose not to allow the autorefs to directly send calls to the teams at all. The presence of a well-founded means for reacting to the feedback and judgments of humans would go a long way toward making the participants of the league feel more comfortable with the autoref and increasing the usage of autorefs in practice, as described below.

[1]At the previous tournament, large computer screens at the fields showed the autoref output.

League-wide adoption We feel that the use of the autoref in the SSL presents, in some ways, a particularly useful arena for testing a robot algorithm. On the positive side, the tournament offers dozens of games during which to run the autoref, each of which is a strong real-world test of the its applicability; the constraint of not displeasing the competing teams too much gives a strong incentive to make sure the autoref works well. Additionally, the TC, if prepared to do so, can decide how to use the autorefs. On the other hand, the tournament only occurs once a year, and realistic testing conditions are hard to come by otherwise.

At recent tournaments, the SSL TC (which included, for the 2017 tournament, the author of this thesis), has made efforts to increase usage of autorefs during competition games; the consensus among both TC members and other participants is, at least in principle, that the league should aim for as much use of autorefs as possible. There is, however, some resistance when team members are faced with the uncertainty of using autorefs in their games, so there is a way to go before the league uses and observes autorefs for every game as a matter of course.

Appendix A

Notation

In this appendix, we define some of the typographic conventions and common quantities used in the pseudocode listings in this thesis.

A.1 Typographic conventions

- *variable-name*: a variable
- *ConstantName*: a constant
- FunctionName: a function
- $\langle x, y \rangle$: a two-dimensional vector, which can represent, e.g., a location, velocity, or acceleration

A.2 Common variables

- t: the current time
- *ball*: the state of the ball in an SSL algorithm

A.3 Common functions

- $\textsc{Sgn}(x)$: the sign of x (i.e. $\textsc{Sgn}(x) = 1$ if $x > 0$ and $\textsc{Sgn}(x) = -1$ if $x < 0$)
- $\textsc{Pos}(x)$: the position of object x
- $\textsc{Vel}(x)$: the velocity of object x
- $\textsc{Distance}(x, y)$: the distance between x and y (which may be objects that have positions or themselves positions)
- $\textsc{x}(v)$, $\textsc{y}(v)$: the x- and y- components of vector v
- $\textsc{Output}(f)$: a function with the side effect of logging the visualization frame f

A.4 Common constants

- *MaxRobotRadius* = 90 mm: the maximum allowed radius of any robot (which we treat as the actual radius of all robots)
- *BallRadius* = 21 mm: the radius of a regulation SSL ball
- *FieldLengthH* = 4500 mm, *FieldWidthH* = 3000 mm: half the length (resp. width) of the field (i.e., the distance from the center of the field to either of the goal (resp. touch) lines; we give the nominal values from the rules here, but any particular field may differ)
- *DefenseRadius* = 1000 mm: the radius of the arcs making up the boundaries of the defense areas
- *GoalWidthH* = 500 mm half of the width of the opening of each goal

A.5 Data structures

- []: an empty list data structure
- $\textsc{Length}(l)$: the number of items in list l
- $x :: l$, $l :: x$: the result of prepending or appending the item x to list l
- $l[n]$: the n^{th} element of list l, using 0-based indexing and with wraparound (e.g. $l[-1] = l[\textsc{Length}(l) - 1]$)

Bibliography

1st & Ten Graphics, 2014. URL https://www.sportvision.com/football/1st-ten%C2%AE-graphics.

Algomation, 2014. URL http://www.algomation.com.

Algoviz, 2009. URL http://www.algoviz.org.

Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, Lecture Notes in Computer Science, pages 209–229. Springer, 1993.

Peter Amstutz and Andrew H. Fagg. Real time visualization of robot state with mobile virtual reality. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 241–247. IEEE, 2002.

Matías Arenas, Javier Ruiz-del Solar, Simón Norambuena, and Sebastián Cubillos. A robot referee for robot soccer. In *RoboCup 2008: Robot Soccer World Cup XII*, pages 426–438. Springer, 2009.

Ronald T. Azuma. A survey of augmented reality. *Presence*, 6(4):355–385, 1997.

Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, Apr 1996. ISSN 0018-9162. doi: 10.1109/2.488299.

Joydeep Biswas, Juan Pablo Mendoza, Danny Zhu, Benjamin Choi, Steven Klee, and Manuela Veloso. Opponent-driven planning and execution for pass, attack, and defense in a multi-robot soccer team. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems*, pages 493–500. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

Gary Bradski. OpenCV. *Dr. Dobb's Journal of Software Tools*, 2000.

Marc H. Brown and Robert Sedgewick. A system for algorithm animation. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pages 177–186. ACM, 1984.

Brett Browning, James Bruce, Michael Bowling, and Manuela Veloso. STP: Skills, tactics, and plays for multi-robot control in adversarial environments. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 219(1):33–52, 2005.

James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2383–2388. IEEE, 2002.

James Bruce and Manuela Veloso. Fast and accurate vision-based pattern detection and identification. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation*, volume 1, pages 1277–1282. IEEE, 2003.

James Bruce, Tucker Balch, and Manuela Veloso. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of the 2000 IEEE/RSJ Conference on Intelligent Robots and Systems*, pages 2061–2066. IEEE/RSJ, October 2000.

Ravi Teja Chadalavada, Henrik Andreasson, Robert Krug, and Achim J. Lilienthal. That's on my mind! robot to human intention communication through on-board projection on shared floor space. In *European Conference on Mobile Robots*, 2015.

Mao Chen, Klaus Dorer, Ehsan Foroughi, Fredrik Heintz, Zhanxiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummenje, Jan Murray, Itsuki Noda, Olivor Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. *RoboCup Soccer Server Users Manual*, 2003. URL http://sourceforge.net/projects/sserver/files/rcssmanual/9-20030211/.

Toby H. J. Collett. *Augmented Reality Visualisation for Mobile Robot Developers*. PhD thesis, University of Auckland, 2007.

SSL Technical Committee. Laws of the RoboCup Small Size League 2015, 2015. URL http://wiki.robocup.org/wiki/File:Small_Size_League_-_Rules_2015.pdf.

Hosam K. Fathy, Zoran S. Filpi, Jonathan Hagena, and Jeffrey L. Stein. Review of hardware-in-the-loop simulation and its prospects in the automotive area. In *Proc. SPIE*, May 2006.

David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice-Hall Englewood Cliffs, 2003.

Michael Friendly and Daniel J. Denis. Milestones in the history of thematic cartography, statistical graphics, and data visualization, 2001. URL http://www.datavis.ca/milestones.

Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

Jean-Bernard Hayet, Justus Piater, and Jacques Verly. Robust incremental rectification of sports video sequences. In *British Machine Vision Conference (BMVC'04)*, pages 687–696, 2004.

Thomas A. Henzinger. The theory of hybrid automata. In M.Kemal Inan and RobertP. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 265–292. Springer Berlin Heidelberg, 2000.

Stanley K. Honey, Richard H. Cavallaro, Jerry Neil Gepner, Edward Gerald Goren, and David Blyth Hill. Method and apparatus for enhancing the broadcast of a live event. US Patent number 5917553 A.

R. Isermann, J. Schaffnit, and S. Sinsel. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice*, 7(5):643–653, May 1999. ISSN 0967-0661.

Uwe-Philipp Käppeler, Oliver Zweigle, Kai Häußermann, Hamid Rajaie, Andreas Tamke, Andreas Koch, Bernd Eckstein, Fabian Aichele, Daniel DiMarco, Adam Berthelot, Thomas Walter, and Paul Levi. RFC Stuttgart team description 2010, 2010. URL ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/INPROC-2010-97/INPROC-2010-97.pdf.

Won S. Kim. Virtual reality calibration and preview/predictive displays for telerobotics. *Presence: Teleoperators and Virtual Environments*, 5(2):173–190, 1996.

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, pages 340–347. ACM, 1997.

Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report 98-11, Iowa State University, 1998.

Juan Pablo Mendoza, Joydeep Biswas, Philip Cooksey, Richard Wang, Steven Klee, Danny Zhu, and Manuela Veloso. Selectively reactive coordination for a team of robot soccer champions. *Proceedings of AAAI-16*, 2016.

Paul Milgram and Herman Colquhoun. A taxonomy of real and virtual world display integration. *Mixed reality: Merging real and virtual worlds*, pages 1–26, Mar 1999.

Paul Milgram and Fumio Kishino. A taxonomy of mixed reality visual displays. *IEICE Transactions on Information and Systems*, 77(12):1321–1329, 1994.

Paul Milgram, Anu Rastogi, and Julius J. Grodski. Telerobotic control using augmented reality. In *IEEE International Workshop on Robot-Human Communication*, pages 21–29, Jul 1995.

Pieter J Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Hybrid Systems: Computation and Control*, pages 165–177. Springer Berlin Heidelberg, 1999.

Tim Niemüller, Gerhard Lakemeyer, Alexander Ferrein, S Reuter, D Ewert, S Jeschke, D Pensky, and Ulrich Karras. RoboCup Logistics League sponored by Festo: A competitive factory automation testbed. In *Proceedings of the 16th International Conference on Advanced Robotics — 1st Workshop on Developments in RoboCup Leagues*, 2013.

Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1992.

Patrick Riley, Peter Stone, and Manuela Veloso. Layered disclosure: Revealing agents' internals. In *Intelligent Agents VII: Agent Theories Architectures and Languages*, pages 61–72. Springer, 2001.

David Robert and Cynthia Breazeal. Blended reality characters. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 359–366. ACM, 2012.

David Robert, Ryan Wistorrt, Jesse Gray, and Cynthia Breazeal. Exploring mixed reality robot gaming. In *Proceedings of the fifth international conference on tangible, embedded, and embodied interaction*, pages 125–128. ACM, 2011.

RoboCup 2014 Technical Challenges. Robocup 2014 technical challenges. URL `http://robocupssl.cpe.ku.ac.th/robocup2014:technical_challenges`.

Robotics Erlangen autoref. URL `https://github.com/robotics-erlangen/autoref`.

Roy J. Rosser, Yi Tan, Howard J. Kennedy, Jr., James L. Jeffers, Darrell S. DiCicco, and Ximin Gong. Image insertion in video streams using a combination of physical sensors and pattern recognition, 2000. US Patent number US6100925 A.

F.B.F. Schoenmakers, G. Koudijs, C.A. Lopez Martinez, M. Briegel, H.H.C.M. van Wesel, J.P.J. Groenen, O. Hendriks, O.F.C. Klooster, R.P.T. Soetens, and M.J.G. van de Molengraft. Tech United Eindhoven team description 2013: Middle Size League, 2013. URL `http://www.techunited.nl/media/files/TDP2013.pdf`.

Jean Serra. *Image analysis and mathematical morphology*. Academic Press, Inc., 1983.

Dan Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

Simspark wiki: Soccer simulation. URL `http://simspark.sourceforge.net/wiki/index.php?title=Soccer_Simulation&oldid=3043`.

ssl-autonomous-refbox. URL `https://code.google.com/p/ssl-autonomous-refbox`.

Ryohei Takahashi, Hiroto Ise, Atsushi Konno, Masaru Uchiyama, and Daisuke Sato. Hybrid simulation of a dual-arm space robot colliding with a floating object. In *IEEE International Conference on Robotics and Automation*, pages 1201–1206, May 2008. doi: 10.1109/ROBOT.2008.4543367.

Kumiko Tanaka-Ishii, Hideyuki Nakashima, Itsuki Noda, Kôiti Hasida, Ian Frank, and Hitoshi Matsubara. MIKE: An automatic commentary system for soccer. In *Proceedings of the International Conference on Multi Agent Systems*, pages 285–292. IEEE, 1998.

Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, 2001.

Douglas Vail, Manuela Veloso, and John Lafferty. Conditional random fields for activity recognition. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, page 235. ACM, 2007.

Manuela Veloso, Nicolas Armstrong-Crews, Sonia Chernova, Elisabeth Crawford, Colin McMillen, Maayan Roth, Douglas Vail, and Stefan Zickler. A team of humanoid game commenters. *International Journal of Humanoid Robotics*, 5(03):457–480, 2008.

Manuela Veloso, Joydeep Biswas, Philip Cooksey, Steven Klee, Juan Pablo Mendoza, Richard Wang, and Danny Zhu. CMDragons 2015 extended team description, 2015.

Visualgo, 2011. URL `http://www.visualgo.net`.

Dirk Voelz, Elisabeth André, Gerd Herzog, and Thomas Rist. Rocco: A RoboCup soccer commentator system. In *RoboCup-98: Robot Soccer World Cup II*, pages 50–60. Springer, 1999.

Stefan Zickler. *Physics-Based Robot Motion Planning in Dynamic Multi-Body Environments*. PhD thesis, Carnegie Mellon University, Thesis Number: CMU-CS-10-115, May 2010.

Stefan Zickler, Tim Laue, Oliver Birbach, Mahisorn Wongphati, and Manuela Veloso. SSL-Vision: The shared vision system for the RoboCup Small Size League. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 425–436. Springer, 2009.