RoboViz: Programmable Visualization for Simulated Soccer

Justin Stoecker and Ubbo Visser

Department of Computer Science
University of Miami, Coral Gables FL
{justin,visser}@cs.miami.edu

Abstract. This work describes RoboViz, a new software program designed to assess and develop agent behaviors in a multi-agent system, the RoboCup Soccer Simulation 3D sub-league. RoboViz is an interactive monitor that renders both agent and world state information in a three-dimensional scene. In addition, RoboViz provides programmable remote drawing functionality to agents or other clients that can communicate over a network. The tool facilitates real-time visualization of agents running concurrently on the SimSpark simulator to provide higher-level analysis of agent behaviors not currently possible with existing tools. Provided appropriate hardware, the monitor and debugging tool can produce high-quality stereo vision images. RoboViz is proposed as a replacement for the current SimSpark 3D league monitor to benefit developers as well as elevate public interest in the 3D simulation league, and it has been used officially at the 2011 German Open in Magdeburg, Germany. RoboViz was released in February 2011 as an open-source project under the Apache 2.0 license.

1 Introduction

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

Presently, there is a lack of generally accessible or effective software tools for analyzing and supporting development of agents for the simulation 3D sub-league. Many researchers in the 3D league are capable of writing specialized programs for their needs, and it is common to see each team develop their own tools. Spending significant time on such specialized tools distracts from the overarching goals of the simulation league; the capabilities and flexibility of team-specific tools are reduced as a consequence. Sifting through the immense amounts of data generated from these simulations remains a challenge. Researchers in the simulation league face a shared set issues while developing agents' reasoning, skills, and behavior; many of these behaviors possess spatial and temporal properties that are amenable to visual presentation.

Compared to the physical leagues, the simulation league also struggles with a less impressive presentation [10]. The current monitor for rendering the simulation has not seen significant improvements over the past years. Some of the commonly mentioned issues with the SimSpark monitor [1] include poor performance, dated graphics, and an awkward user interface. The simulation league in RoboCup is a useful platform to pioneer and experiment without physically risking robots; this league should be exciting and at the forefront of research before it is deployed in the physical leagues. The existing monitor used to view the simulations does not reflect the state of the art research being conducted in this field, and for the simulation league to attract increased public interest the monitor must be updated accordingly.

This paper proposes that there is an unmet potential for high-quality analysis and development of agents and their algorithms in the 3D simulation league through real-time visualization. This is supported by a software solution that fills the role of a simulation monitor with visual data overlaid and integrated with the 3D scene. Our solution is designed to be accessible to resolve the shared debugging and analysis issues researchers encounter. This program addresses both the visualization needs of developers while providing a significant upgrade over the existing monitor to benefit the community as a whole.

2 Related Work

Several teams competing in the simulation leagues of RoboCup develop their own tools to optimize and debug their agent behaviors. For the most part, these tools are described in team description papers. The 2D simulation team *Mainz Rolling Brains* developed a debug and visualization tool called *FUNSSEL* [3]. *FUNSSEL* acts as a layer between the server and agents to intercept and process communication. The primary features of this software include filtering data, agent training, and graphic overlays for the 2D field in a secondary monitor. Other examples of tools for the 2D league can be found, for example, in the Portuguese team *FC Portugal* [9] and many other team description papers not mentioned here for brevity. For the 3D simulation league, the team *Virtual Werder 3D* utilized an evaluation tool [5] to analyze agent performance. The program also supported basic drawings in a simplified 2D monitor; however, all analysis and visualization was done on server and agent generated logfiles.

There have been few attempts at providing useful analysis and debugging tools for the general community. The *logfile player and analyzer* [8] provided improvements to the log playing capabilities of the 3D monitor; for example, it allows agents to record behaviors as simple drawings displayed in the 3D scene when the log is replayed. Additional features of this program included some basic filtering of logfile data and an improved graphical user interface.

Simulators for multi-agent systems often include some manner of visualization. Usually, these visualizations refer to modeling the robots and environment.

However, a few simulators do provide additional functionality. In Webots [6], for example, user code can initiate the drawing of primitives to further model the robot components. Breve [4] is another simulation program that supports the modeling and simulation of large multi-agent environments. Breve also exposes simple drawing routines to add shapes to the scene.

Despite the importance of processing, visualizing, and understanding data output from 3D league simulations, there is a dearth of solutions to address these needs. This is evidenced by the scarcity of high-quality tools available to the 3D simulation league community. Tools mentioned in team description papers are, unfortunately, not well documented, obsolete, or unsuitable for general use as they may be tightly bound to a team's agent architecture. Simulators such as Webots and Breve, while providing more advanced visualization capabilities, are primarily focused on the simulations themselves. The drawing routines exposed by these simulators are secondary and not integrated with the interface in a meaningful way. The existing SimSpark monitor [1] has a number of limitations as well. In particular, we felt the following issues should be resolved or improved upon:

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

The current 3D monitor fulfills the basic requirement that it is real-time, and it is also marginally interactive by allowing users to control a virtual camera; however, the extent of its visualization capabilities is limited to presenting the physical world state, or ground truth. The SimSpark monitor is functional merely as a passive viewer, and as a visualization tool it is incapable of conveying information other than actual positions of players and the ball.

3 Requirements

Before designing RoboViz, we looked at the needs of developers and the problems with current solutions. The most crucial issue we looked into was determining how researchers view and process data from the simulations. Teams in the 3D simulation league typically output values specific to their implementation to the terminal or a file. This form of data can be used for correcting the underlying algorithms when the output doesn't match expectations, and the code required is minimal. The obvious problem with text is that it does not provide any higher-level understanding of behaviors; by itself, textual data does not reveal patterns or other interesting situations that may lead to unexpected results. A deeper analysis is only achievable by processing this data, which requires further tools of some kind.

Given the nature of the soccer simulation, a greater amount of data is geometric and spatial and better suited to a visual representation. A large portion of research efforts may also be spent on algorithms that are inherently spatial or temporal: path planning, localization, and obstacle avoidance are a few examples. Visualization is a natural choice for understanding and debugging these types of algorithms. The aforementioned *logfile player and analyzer* attempted to provide a visual overlay or drawing component for analyzing logfiles; however, the implementation was crude and limited to logged games. The following observations may be made concerning the requirements for effectively visualizing the general scenario of soccer-playing robots:

– Logging data and analyzing it post-simulation is useful, but it is better to possess real-time information. An online system allows researchers to influence and analyze behaviors interactively, which is essential in testing various cases.
– Agents are not omniscient, and their belief state is just as relevant as their actual state. The visualization must be capable of presenting both forms of information simultaneously and effectively.
– It is ideal to abstract the visualization in such a way that is suitable for multiple agent architectures. However, there should be no compromise in functionality to achieve this. The visualization should be easily programmable and flexible.
– Agent behaviors are reliant upon the state of the environment as much as the decision-making architecture. Viewing a single agent's status, while useful, may not be sufficient to understand why a particular action was performed. It is necessary to provide multiple perspectives of the simulation.

The above observations indicate that a suitable visualization solution runs in real-time with the Simspark simulation, is interactive, and flexible in its presentation of data. We also wanted a tool that has few dependencies, can be setup with minimal effort, and can be used on many operating systems.

4 Approach

While the Simspark monitor does not fulfill the requirements for a visualization tool, it is effective in illustrating the true world model of the simulation. Initially there was no intention of reproducing this functionality, and it was expected that a visualization tool could independently complement the monitor. The earliest prototype for RoboViz was a program detached entirely from the Simspark framework; it required a tight coupling with an agent architecture that supplied all the data needed for visualization. This was done with the belief that a debug tool should not have any dependency on the Simspark framework, which was likely to change over time. Since the agent code would need to be updated to conform to an updated Simspark simulation, this would centralize code modification to the agents.

There are serious drawbacks to the previously mentioned design that appeared during early prototyping. Unless the agent architecture has access to the world model of the simulation, there is no way to visualize both believed and actual world models simultaneously. Viewing the separate models side-by-side, using both the monitor and visualization tool concurrently, is ineffective in situations where there are discrepancies between what an agent believes and the truth. Furthermore, such a design requires much more effort on the part of a team hoping to utilize the visualization features with this interface. These are the reasons that necessitated a revised design.

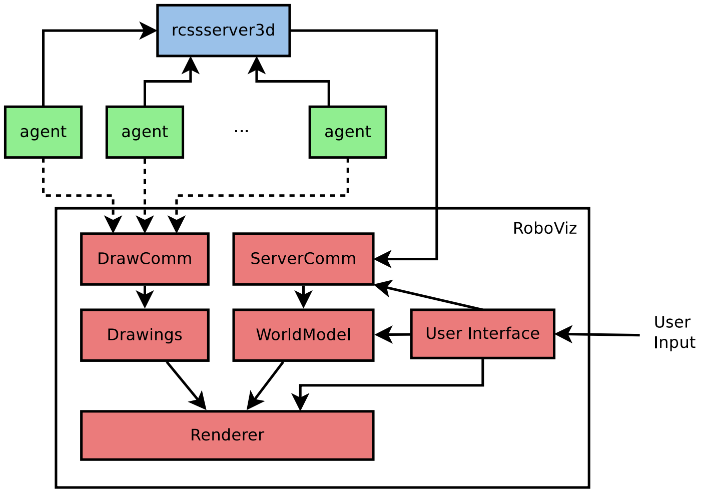


Fig. 1. Architecture for RoboViz, SimSpark server (rcssserver3d), agent, and user interaction

It was concluded that RoboViz would still need to communicate with agents to access their internal states, but it must also use the simulation server's scene graph to render the actual world model. With such a design, RoboViz essentially performs the roles of both the 3D monitor and a visualization tool (Fig. 1). This approach provided an opportunity to address many of the SimSpark monitor's deficiencies.

4.1 Visualization

To facilitate support for multiple agent architectures, it is better to assign the task of generating useful debug information to the agents themselves. The more traditional approach for visualization software is for agents to expose their state, and have the debug tool poll this information and render it appropriately. However, this approach tends to split development into two tasks: developing agent behaviors and then reconfiguring the debug tool to visualize it. This generates unnecessary work, and can easily introduce additional complexity in an environment where multiple individuals are working on different modules of the agent.

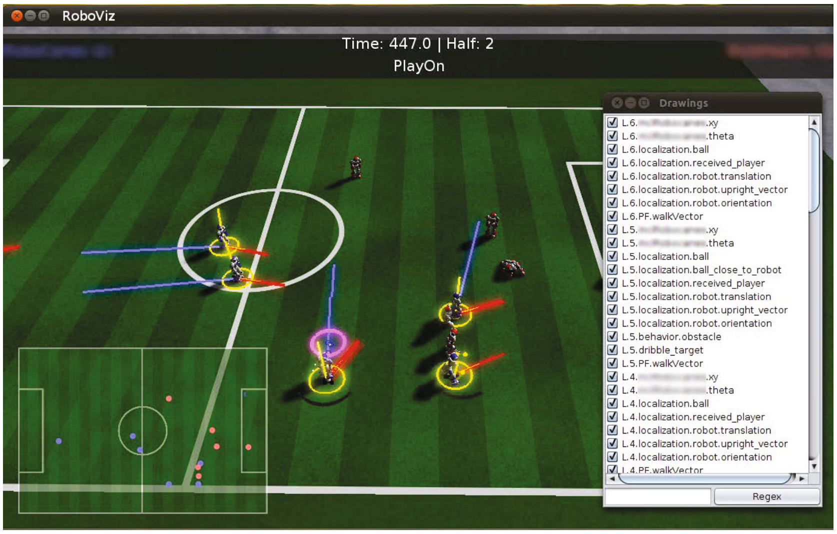
A preferred approach is to delegate the responsibility of producing useful visualizations to the agent. With this approach, the visualization tool acts as a passive receiver and simply renders what the agents would like displayed. Agents have the option of adding several simple shapes, such as line segments, spheres, and points, to the rendering. Each shape can be configured to have a different color, size, thickness, and is also tagged with a name so it can be identified. This makes for a very small and simple interface for agents to use, and more complicated visualizations can be achieved using these basic shapes. This system enables multiple programmers to visualize the algorithms they are working on without needing to reprogram the visualization tool or spend more time writing their own debug program for an individual problem.

One issue with allowing agents to be in control of what is rendered is that users should ultimately decide what they are seeing. Each agent may be sending localization information, strategy decisions, believed opponent locations, and so forth. Having all of this information presented at the same time can be overwhelming, and it can be useful to isolate drawings. This issue can be addressed by allowing the user to selectively filter content agents are providing. Filtering is done using regular expressions on drawing names or toggling individual drawings' visibility.

5 Implementation

RoboViz is a real-time 3D application that connects with the Simspark server using its network protocol and has a simple drawing interface to interface with agents . It relies entirely upon communication over a network (Fig. 1) to process information from live simulations. RoboViz possesses all of the existing functionality of the SimSpark monitor while providing the visualization described in section 4.1. Some of the other features that the program implements:

– *Graphics*: Advanced visual effects including variance shadow mapping [2] and bloom post-processing have been added to improve overall presentation quality. Stereoscopic 3D rendering is also supported on systems with the necessary hardware.
– *Controls*: Enhanced interactivity allows users to move the ball or agents with the mouse, and the user interface includes more intuitive controls.
– *Camera*: In addition to the standard user-controlled camera, there is an automated camera for tracking the ball as it moves around the field. Users may also view the scene from the perspective individual agents.
– *Logging*: RoboViz now takes over the role of generating logfiles, freeing valuable resources for the machine running simulation server. These logs are also



- ✓ L.6.[redacted].xy
- ✓ L.6.[redacted].theta
- ✓ L.6.localization.ball
- ✓ L.6.localization.received_player
- ✓ L.6.localization.robot.translation
- ✓ L.6.localization.robot.upright_vector
- ✓ L.6.localization.robot.orientation
- ✓ L.6.PF.walkVector
- ✓ L.5.[redacted].xy
- ✓ L.5.[redacted].theta
- ✓ L.5.localization.ball
- ✓ L.5.localization.ball_close_to_robot
- ✓ L.5.localization.received_player
- ✓ L.5.localization.robot.translation
- ✓ L.5.localization.robot.upright_vector
- ✓ L.5.localization.robot.orientation
- ✓ L.5.behavior.obstacle
- ✓ L.5.dribble_target
- ✓ L.5.PF.walkVector
- ✓ L.4.[redacted].xy
- ✓ L.4.[redacted].theta
- ✓ L.4.localization.ball
- ✓ L.4.localization.received_player
- ✓ L.4.localization.robot.translation
- ✓ L.4.localization.robot.upright_vector
- ✓ L.4.localization.robot.orientation
- ✓ L.4.PF.walkVector

Fig. 2. The current user interface includes the same game state information as the 3D monitor; however, it also has an optional 2D aerial view, a panel for filtering drawings, an agent-perspective camera

recorded at a greater framerate than the server currently generates logs, resulting in much higher quality logfiles.

– *Interface*: A 2D top-down view is provided as an ancillary means of keeping track of player locations, and a panel is provided to filter visualizations using check-boxes or a regular expression.

As mentioned in section 4, another goal is to achieve a program that is lightweight, configurable, and cross-platform. RoboViz is implemented entirely as a Java application to meet these needs. No external libraries or packages are used except for Java Bindings for OpenGL (JOGL) [7], which provides access to OpenGL rendering. This makes RoboViz especially easy to deploy on any platform, and the source code can be compiled on any Linux, Windows, or OS X system that has Java and recent graphics drivers.

5.1 Drawing Protocol

One of the primary functions of RoboViz is allowing other processes or clients to perform drawing of simple shapes inside of RoboViz. This functionality is referred to as *remote drawing*, and serves as the process by which visualization is achieved. To accomplish *remote drawing*, RoboViz has a simple network protocol for allowing clients to control custom drawing. The primary intention for this is to allow robots running on the SimSpark server to send information useful in debugging their internal state or behavior.

5.2 Commands

Clients interact with RoboViz by issuing commands. Agents submit packets in the format of commands that RoboViz can recognize. For drawing purposes, an example of a command is *draw line*, or *draw sphere*. Each command is formatted in a specific way that is suitable for it to be sent using UDP packets over a network (Fig. 3). We chose UDP primarily because we expect for RoboViz to be run on the same host or local area network (where packet loss is highly uncommon) as the agents and simulation. UDP also simplifies the connections between RoboViz and agents, which may crash unexpectedly.

Each packet may contain one or more commands, though the maximum size of a packet containing commands is currently 512 bytes; a drawing command is typically around 50 bytes, depending on the length of its name. Each command has a specific format (Fig. 3) so that its parameters can be identified unambiguously, even if it is grouped in the same packet with other commands. Commands should not be split between two or more packets. The most important command our visualization and debugging tool recognizes is the *draw shape* command. This command appends a primitive shape to a list of drawings in RoboViz and can be used by individual agents to visualize a particular algorithm in action: locations of opponents, believed world state information, and so forth.

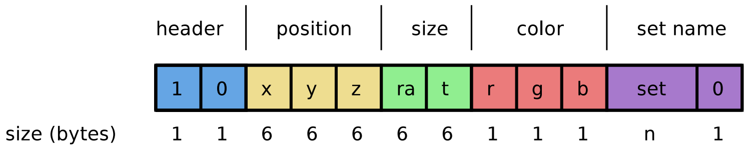


Fig. 3. Example of a *draw shape* command. The header section tells RoboViz the type of command it should parse; the first byte indicates a *draw command*, and the second indicates the shape is a line. For a line, the parameters include position (x,y,z), size (radius, thickness), color (r,g,b), and set name. All floats are stored as 6 bytes of text (ex. “3.1456”). Strings, such as the set name, are terminated with a 0 byte.

5.3 Shapes and Sets

To make remote drawing simple and flexible, all drawing is accomplished by working with a small set of shapes. More complex shapes can be constructed from these primitives, so we avoid introducing an overly verbose drawing interface by permitting only basic shapes. A shape’s description has properties such as position, color, and scaling. Clients send commands to RoboViz that request it to add such shapes to its rendering; these commands are called *draw shape* commands.

Each shape is also part of a group of shapes called a *shape set*; these sets can contain one or more shapes. Every *shape set* is identified by a unique name, which is a string. When RoboViz receives *draw shape* commands from the client, it parses the individual shapes and adds it to the appropriate set. During rendering in RoboViz, each set is visited and its shapes are drawn.

Shapes are grouped into sets mainly so they can be filtered inside RoboViz. For example, each robot may have its own set of shapes so that the user of RoboViz can view only that agent's shapes and hide everything else. A robot may also have many sets for each type of behavior. How the sets are organized is entirely up to the client. However, it is recommended to use a hierarchical naming pattern. For instance, a shape set name such as "TeamName.1.Behaviors.PathPlanning" might be used to indicate all shapes belonging to the first agent on a team called "TeamName" that pertain to path planning behavior. With this naming pattern, all of this particular agent's sets can be referenced as "TeamName.1".

A unique class of shapes are *annotations*, which are text billboards displayed inside the 3D scene. There are two types of *annotations*: positional annotations are strings rendered at a specific location in the scene; agent annotations are attached to a specific agent in the simulation, and always rendered over the agent's head (Fig. 4).

5.4 Rendering Control

While *draw shape* commands request that RoboViz add shapes, they are not enough to sufficiently control the remote drawing process. Commands that influence the rendering process of shapes are called *draw option* commands. These commands are used for achieving animation and resolving concurrency issues.

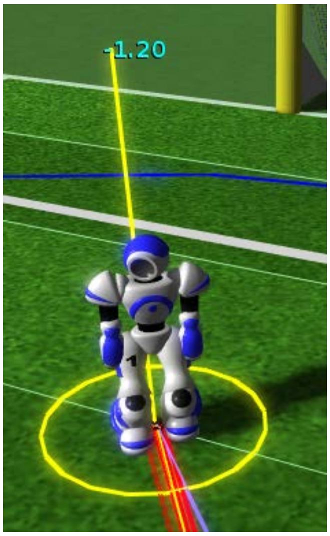


Fig. 4. An agent annotation that displays a numeric value overhead of a goal keeper. Also shown is an upright vector and localization information as a line and circle, respectively.

Problems arise if a shape set is rendered at the same time RoboViz is receiving new *draw shape* commands for that set. To resolve this problem, each shape set has two buffers: a front and back buffer. When a client sends a *draw shape* command, RoboViz parses the shape and add it to the back buffer. These shapes will not be visible until the client sends a *swap buffer* command, at which point the back and front buffers swap roles. In other words, the front buffer always contains a set's shapes which may be rendered in RoboViz. The *swap buffer* command is synchronized inside of RoboViz, and will block while shapes are being rendered.

It is important to note that as soon as RoboViz executes a buffer swap, the shape set's new back buffer is cleared of all shapes. If multiple agents try to swap the buffers of the same shape set, flickering will occur in RoboViz. For this reason it is expected that agents will submit drawings relevant to their own state or behavior within their own shape sets, but this is not strictly required. A team may, for instance, designate a captain that sends drawings relevant to the team as a whole. We did not wish to impose a set of rules on how drawings should be assigned to RoboViz, so there is nothing to prevent agents from sending drawings identified by names other than their own.

5.5 Static and Animated Shapes

The type of shapes a client can add to RoboViz may be categorized into two types: static and animated. Static shapes are those that persist after being added and do not need to be refreshed or updated. For example, a grid on top of the field or 3D coordinate axes do not require the shapes to change over time. Animated shapes are those that need to have their properties updated; a vector representing a robot's forward direction is an example.

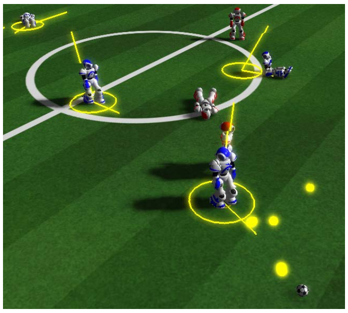
Inside *draw shape* commands, there is absolutely no distinction between static and animated shapes. These types merely refer to how a client program treats the *draw commands* for a shape set. Static shapes have the advantage of only needing to be transmitted a single time and only require a single swap buffer command. Animated shapes must have their values sent repeatedly; each time values are updated the shape set must also have its buffers swapped to display the changes.

6 Results and Future Work

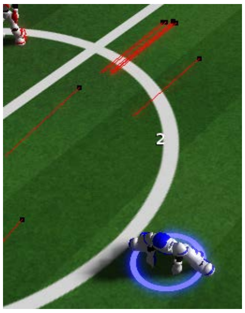
While RoboViz is still being developed, it has been successfully received by the simulation 3D community. The tool has generated a great deal of interest from several teams, and has been suggested as a replacement for the current 3D monitor in the 2011 RoboCup in Istanbul, Turkey; RoboViz was used as the official monitor during the 2011 German Open in Magdeburg. We have not only seen the tool adopted by other teams, but also developers. For instance, the TinMan [11] framework has integrated support for the visualizations. Finally, RoboViz has been used at the University of Miami as a demonstration tool to promote awareness of the RoboCup events.

Applications: We have used the tool to iron out many bugs and optimize several of our agent behaviors. In particular, the tool has been of great use in visualizing our agents' localization routines, path planning, decision-making, and behavior modeling. We expect the usefulness of this tool to expand as we continue to refine existing features and develop our agents. This section provides only a few samples to illustrate how the RoboViz tool can be used to visualize various algorithms and routines.

Localization concerns an agent's knowledge of where it is positioned in a the environment. An agent will have a difficult time with formations, kicking the ball, or avoiding obstacles if it thinks it is in a location it is not. This is one of the most basic and obvious things to visualize, and we represent an agent's believed position as a yellow circle at its base that is drawn just atop the field (see Fig. 5). We also include each agent's perceived upright vector and a sphere where

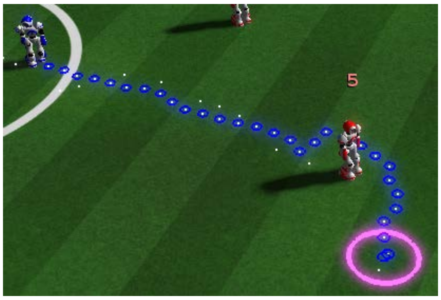


(a)

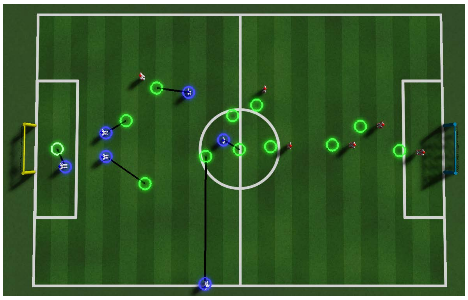


(b)

Fig. 5. Localization: In **(a)**, each agent's upright vector and position are represented by lines and circles, respectively. In **(b)**, individual particles from Monte Carlo localization are seen, scattered around, after an agent is moved unexpectedly. The speed with which these particles converge at the agent's base indicates how quickly it has determined its position.



(a) **Path planning:** A single agent's path plan is laid out as a series of blue circles; the white points indicate other considered routes. In this example, the agent anticipates a collision with an opponent player and schedules the path accordingly.



(b) **Decision making:** Potential tasks, such as field positioning, are illustrated by green circles on the field. Agents are matched with tasks, and their allocated assignments are visualized as connecting black line segments.

Fig. 6.

the agent thinks the ball is. This information can be quickly used to gauge the effectiveness and accuracy of the localization routines and explain unexpected behavior.

Another type of behavior that is well-suited to this type of visualization is path planning and obstacle avoidance. This can be directly visualized by the vertices on an agent's path (see Fig. 6(a)). Some other behaviors, such as allocating tasks to agents, is not as immediately obvious to visualize. We show a basic example of player positioning, where choices and assignments are drawn as connected nodes of a graph (Fig. 6(b)). However, a more sophisticated visualization might include weighting tasks by importance using color or size.

Future Work: There are a few areas where RoboViz is being improved. While a greater emphasis is placed on real-time visualization, analysis capabilities (such as game statistics) may be provided to provide users with more information. Furthermore, drawings are not logged and cannot be used while RoboViz is playing a logfile. Another area that we intend to improve is the user interface and viewing functionality. Currently, RoboViz retains a camera system that is similar to the SimSpark monitor to avoid a learning curve for users. A semi-automated camera mode is available that tracks the ball around the field; ultimately, a more intelligent and fully automated camera would be useful for presenting matches. Finally, the drawing panel that allows users to filter shapes needs to be integrated directly into the main window.

References

1. Boedecker, J., Dorer, K., Rollmann, M., Xu, Y., Xue, F.: SimSpark User's Manual (June 2008)
2. Donnelly, W., Lauritzen, A.: Variance shadow maps. In: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D), pp. 161–165. ACM Press (2006)
3. Arnold, A., Flentge, F., Schneider, C., Schwandtner, G., Uthmann, T., Wache, M.: Team Description Mainz Rolling Brains 2001. In: Birk, A., Coradeschi, S., Tadokoro, S. (eds.) RoboCup 2001. LNCS (LNAI), vol. 2377, pp. 531–534. Springer, Heidelberg (2002)
4. Klein, J., Spector, L.: 3D Multi-Agent Simulations in the breve Simulation Environment. In: Komosinski, M., Adamatzky, A. (eds.) Artificial Life Models in Software, pp. 79–106. Springer, London (2009)
5. Lattner, A.D., Rachuy, C., Stahlbock, A., Warden, T., Visser, U.: Virtual Werder 3D Team Documentation 2006. Tech. Rep. 36, TZI, Universitaet Bremen (August 2006)
6. Michel, O.: Webots: Professional Mobile Robot Simulation. Journal of Advanced Robotics Systems 1(1), 39–42 (2004)
7. Java Bindings for OpenGL (JOGL), `http://www.jogamp.org`
8. Planthaber, S., Visser, U.: Logfile Player and Analyzer for RoboCup 3D Simulation. In: Lakemeyer, G., Sklar, E., Sorrenti, D.G., Takahashi, T. (eds.) RoboCup 2006. LNCS (LNAI), vol. 4434, pp. 426–433. Springer, Heidelberg (2007)
9. Reis, L.P., Lau, N.: FC Portugal Team Description: RoboCup 2000 Simulation League Champion. In: Stone, P., Balch, T., Kraetzschmar, G.K. (eds.) RoboCup 2000. LNCS (LNAI), vol. 2019, pp. 29–40. Springer, Heidelberg (2001)
10. Shahri, A.H., Monfared, A.A., Elahi, M.: A Deeper Look at 3D Soccer Simulations. In: Visser, U., Ribeiro, F., Ohashi, T., Dellaert, F. (eds.) RoboCup 2007. LNCS (LNAI), vol. 5001, pp. 294–301. Springer, Heidelberg (2008)
11. TinMan: c-Sharp framework for 3D simulation league, `http://code.google.com/p/tin-man/`