Visualizing and Debugging Complex Multi-Agent Soccer Scenes in Real Time

Justin Stoecker and Ubbo Visser

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

Abstract. The RoboCup Soccer environment is one of the most difficult scenarios for autonomous agents. With the potential for so many things to go wrong, debugging and analyzing agents' behaviors becomes a significant task. We propose RoboViz, an open-source program for integrating agent-driven visualizations into a real-time, 3D rendered environment; the scene becomes a shared, interactive whiteboard for all agents, and the user can moderate by filtering drawings they are interested in. Visualization is an effective tool for tracking down errant behaviors and explaining algorithms. RoboViz is embraced by the RoboCup Soccer Simulation 3D sub-league as the de facto monitor application, and the latest revision makes it useful for other leagues as well. We are currently testing RoboViz in the Standard Platform League (SPL).

1 Introduction

A significant challenge in developing a robotic agent is debugging and evaluating its behavior. The RoboCup Soccer scenario is one of the most difficult environments for intelligent agents, and presents several hurdles: an uncertain and dynamic world, multiple competitive and cooperative agents, physics, and the need for high-level strategy. With such a challenging environment, it is inevitable that teams developing soccer agents for RoboCup will stumble over bugs and struggle with solutions. As might be expected, teams competing in the various RoboCup leagues develop their own specialized tools, in isolation, to optimize and debug agent code. However, there is generally a lack of tools that are truly beneficial to all teams participating in a league. In this paper, we focus on the development and debugging issues shared by teams in the RoboCup Soccer Simulation 3D sub-league; however, the same issues are relevant to other RoboCup Soccer sub-leagues.

There are several challenges to overcome when interpreting robot behaviors, such as localization or task planning, and simple approaches are often inadequate in diagnosing problems. The real-time nature of the environment is the most notable complication: agents generally process input and act within milliseconds, so outputting values on a console provides an incomprehensible amount of information. Logging this information and parsing it later has two serious drawbacks: the data can fill volumes very quickly, and it can be difficult to synchronize one agent's

S. Behnke et al. (Eds.): RoboCup 2013, LNAI 8371, pp. 640–647, 2014.

view of the world with an external ground truth. Because of the dynamic environment, inspecting a single agent using an interactive debugger, such as the GNU Debugger, can be disruptive to a simulation or game and change the outcome.

Our answer to these problems is to visually integrate agent state and behavior information into a 3D representation of the environment. The design of these visualizations is delegated to individual agents; the soccer scene becomes a shared whiteboard for all agents, and the user can moderate by filtering drawings they are interested in. The primary contribution we present is a software program, called RoboViz, which enables real-time visualization that is accessible through a simple drawing protocol. RoboViz is also an implementation of the SimSpark [2] soccer simulation monitor, and it provides additional functionality not previously available. We expand on our recent work in [10] by redesigning the software to be usable in environments outside 3D simulation.

In the next section, we cover some of the previous work that has been done with regard to debugging or visualization in the RoboCup Soccer scenario. In section 3, we describe our visualization approach. We comment on how RoboViz has affected the 3D simulation sub-league in section 4, and we end with some ideas for future progression and applicability to other leagues and areas.

2 Related Work

Several teams competing in the simulation leagues of RoboCup develop their own tools specific to their team's agent architecture. 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* [1]. *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 logs.

There have been few attempts at providing useful analysis and debugging tools for the 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 are for modeling the robots and environment. However, a few simulators do provide additional functionality. In Webots [7], 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. Yet another simulator that has been used in the SPL would be SimRobot [6].

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 particular agent framework. Simulators such as Webots and Breve, while providing more advanced visualization capabilities, are unwieldy or impossible to integrate into the SimSpark simulation; additionally, the drawing routines exposed by these simulators are secondary and not integrated with the interface in a meaningful way.

3 Approach

We frame our technical solution to visualizing agent state and behavior by using the RoboCup Soccer Simulation 3D sub-league as an example. Other environments, such as the Standard Platform League, are less complicated to work with: there is no server component to interface with. Games in the RoboCup Soccer Simulation 3D sub-league consists of three components: (1) the simulation server, implemented by *rcssserver3d*, orchestrates the simulation by computing physics, enforcing rules, and managing the update cycle; (2) agents connect to the simulation server to receive *perceptors* (sensor input and game state changes) and send *effectors* (desired joint changes); (3) a monitor, implemented by *rcssmonitor3d*, communicates with the simulation server to receive a *scene graph* for rendering, and allows a user to referee the game. The simulation server, monitor, and each agent runs as an individual processes that communicate using TCP and UDP sockets.

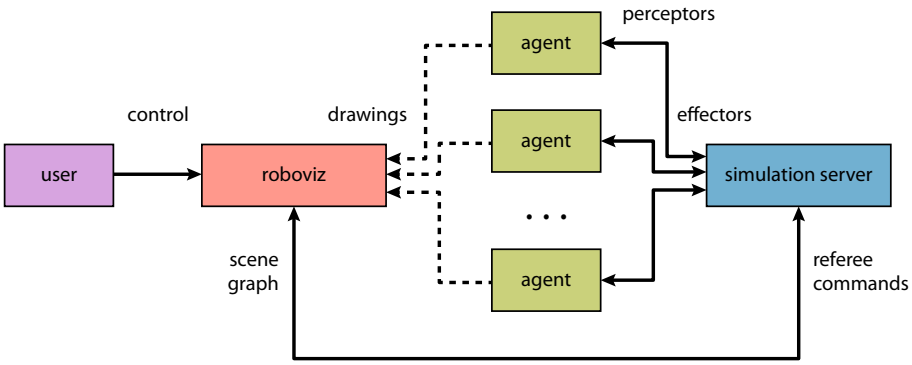


Fig. 1. Interaction between the components of a simulation-3D game, with RoboViz replacing the default monitor. Each component runs as a separate process, with communication handled by either TCP or UDP sockets. The dotted arrows between agents and RoboViz indicate the optional visualization commands.

RoboViz is our implementation of an advanced monitor, and it effectively replaces the default implementation provided by *rcssmonitor3d*. The simulation server communicates the environment in *scene graph* and *game state* objects; these objects are sent to any monitor connected to the server when a change occurs. The scene graph stores only the information necessary to render the environment: geometric primitives, model names, transform matrices, lights, and so forth. The game state contains information about the time, scores, team names, rules, play mode, and field dimensions. It is up to the monitor to decide how to parse this information and display it to a user. The monitor can also send messages to the simulation server, which is useful for referee purposes.

In the physical sub-leagues, there is (clearly) no server to provide a scene graph. The 3D scene must be rendered by a module in RoboViz that is specific to the environment. For example, a Standard Platform League module is implemented that draws a field, field lines, and goals according to the dimensions outlined in the rules. These modules require much less effort to implement because there is no network communication or parsing required; these scenes contain mostly static geometry, and the agents can visualize themselves using the drawing protocol which is discussed next.

Efficiently rendering a scene is a non-trivial task, and RoboViz provides modernized graphics in comparison to *rcssmonitor3d*; however, the rest of this paper will focus on the agent-driven visualization capabilities of RoboViz. A protocol is provided that allows external processes to submit 3D drawings, in real time, that are integrated directly into soccer scene. Figure 2 shows an example where agents on the blue team are drawing circles, lines, and points that represent believed orientations, path-planning, localization particles, and so forth.

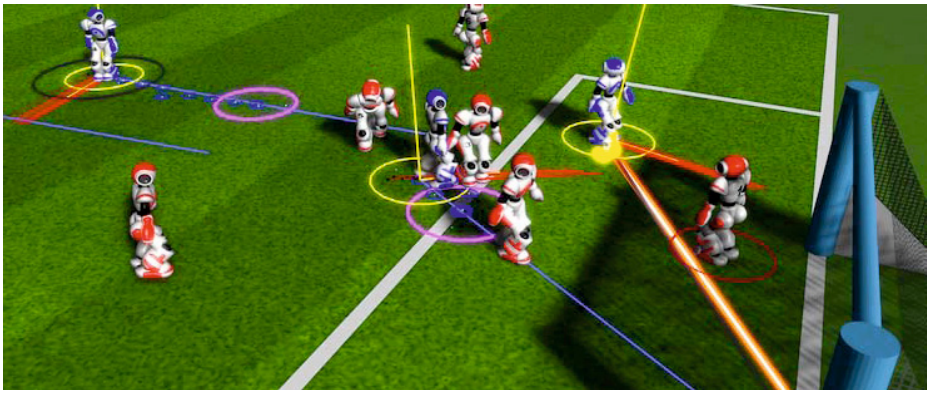


Fig. 2. This rendered scene highlights some of the graphics and visualization capabilities of RoboViz. The robots, field, ball, and goals are constructed entirely from the simulation server's scene graph message. Colored drawings, like the circles and lines, are shapes submitted directly from individual agents in the RoboCup Simulation 3D sub-league.

3.1 Drawing Protocol

A simple network protocol enables clients, such as agents, to draw shapes in RoboViz. This visualization interface is intentionally low level, and it provides great flexibility and is easy to integrate into existing agent architectures. Clients interact with RoboViz by issuing commands. An example of a command is *draw line*. Commands are serialized into bytes[1], and each command has a unique format so it can be tightly packed in a buffer. When a client is ready to submit drawings, it can fire off a UDP packet to RoboViz that contains one or more commands. We chose UDP primarily because we expect for RoboViz and agents to be run on the same host or LAN, where packet loss is highly uncommon. UDP simplifies the connections between RoboViz and agents, which may crash unexpectedly, and makes it easier for teams to add drawing functionality to their agents.

To make the visualization interface flexible, all drawing is accomplished by working with a small set of shapes: circles, points, lines, spheres, and convex polygons. More complex shapes can be constructed from these primitives. In addition to geometric shapes, strings can be rendered by RoboViz. We refer to both geometric shapes and text drawings as *draw commands*. Draw commands includes properties such as position, color, and scaling, and they also contain a name.

The name assigned to a drawing is very important, because all drawings with the same name will be grouped into a *shape set* within RoboViz. Grouping shapes into sets is useful so drawings can be filtered inside RoboViz. For example, each robot may have its own set of shapes, so the user can choose to render only that particular agent's shapes and hide everything else. A robot may also have sets based on behaviors or algorithms. How the shape names are chosen is entirely up to the agent programmer; however, we recommend a hierarchical naming pattern, as the shapes are stored as a tree in RoboViz. Each shape set is a node in the tree. For instance, a shape set name such as *RoboCanes.1.PathPlanning* could be used to indicate all shapes belonging to the first agent on the RoboCanes team that pertain to path planning; this shape set is the child of *RoboCanes.1*, which is the child of *RoboCanes*. The advantage of this convention is that the user can filter rendered drawings very easily.

3.2 Rendering Control

Let's say an agent wants to draw a circle to represent its current location. This position changes constantly, so a new circle is drawn every cycle. Very quickly, the scene is flooded with circles because none of the old ones have been removed. The drawing protocol has another command, *swap buffers*, that will clear out all of the existing shapes whose names begin with a given string. For example, the shape sets *RoboCanes.1.PathPlanning* and *RoboCanes.1.Localization* would

[1] For detailed command formats refer to the documentation:
`https://sites.google.com/site/umroboviz/drawing-api/draw-commands`

both be cleared by issuing a *swap buffer* with *RoboCanes.1* as the argument. This allows the programmer to avoid lots of extra commands if they organize their shape sets in a hierarchical fashion.

The command is called *swap buffers* instead of *clear* or *reset* because it addresses another problem. During rendering, it's not possible for RoboViz to iterate over a set of shapes while new shapes are being added: the rendering and network communication of RoboViz are handled by separate threads. Each shape set has two buffers: a front and back buffer. When a client sends a draw command, RoboViz parses the shape and adds it to the back buffer. These shapes will not be visible until the client sends a swap command, at which point the back and front buffers exchange roles. This behavior enables asynchronous reading and writing of shapes. In other words, the front buffer always contains a set's shapes which may be rendered in RoboViz. The *swap buffers* 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.

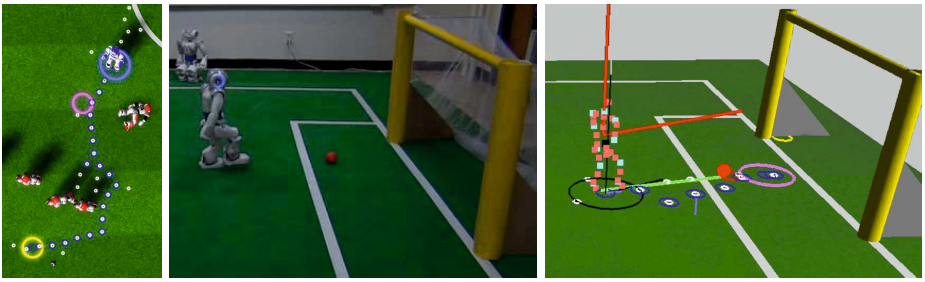
3.3 Implementation

RoboViz was programmed in Java to mitigate cross-platform issues and reduce the number of libraries needed: the Java runtime environment provides solutions for networking, threading, windowing, and GUI development. 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. Rendering is done entirely with OpenGL through the Java Bindings for OpenGL (JOGL) library [3]. We have made RoboViz open source under the Apache 2.0 License.

4 Results

RoboViz is particularly useful for visualizing belief states and high-level strategies. We have seen drawings for localization routines, path planning, decision-making, and behavior modeling. Figure 3 shows visualizations from robots in the Simulation 3D and SPL environments.

RoboViz has been adopted as a monitor for the RoboCup Soccer Simulation 3D sub-league since it was announced: the source code was released in February 2011, and its first public use in competition was at the 2011 RoboCup German Open held in Magdeburg, Germany. This was followed by official use at



(a) (b) (c)

Fig. 3. In (a), a Sim3D agent's search tree is visualized to show path planning from the blue circle (current state) to the yellow circle (desired state). The chosen path is highlighted in blue, with other options in white, and the pink circle is the first destination to reach without collision. In (b), a NAO robot kicks a ball towards the goal; this scene is visualized in RoboViz in (c): the red and blue dots indicate the robot part centers and orientations; the red vectors are the robot's orientation; and the blue circles indicate the path.

RoboCup 2011 in Istanbul, Turkey and RoboCup 2012 in Mexico City, Mexico. The visualization feature of RoboViz has been popular with many teams in the 3D simulation sub-league. In particular, we would like to note that many of the teams in the RoboCup 2012 semi-finals used RoboViz during development. We have also seen agent frameworks released that explicitly provide support for RoboViz [11].

Outside of RoboCup, we have seen RoboViz incorporated into courses in multi-agent systems at the University of Miami and the University of Texas at Austin[2]. RoboViz has also been used at the University of Miami as a demonstration tool to promote awareness of the RoboCup events and attract students to computer science.

5 Conclusion and Future Work

Our initial goal with RoboViz was to present a tool that was accessible and useful to the entire simulation 3D sub-league. We believe this has been clearly demonstrated over the past two years: several teams make use of the visualization capabilities, and RoboViz is used at the major competitions as the monitor. As a bonus, RoboViz has been used for education purposes outside of RoboCup.

Moving forward, we would like to see RoboViz used in other RoboCup soccer leagues, such as the Standard Platform League. It would be especially interesting to see teams that write cross-league agents that can reuse the same tools. While the primary release of RoboViz[3] is still focused on the Simulation 3D sub-league, a branch[4] is under development that makes the tool usable for many other

[2] `http://www.cs.utexas.edu/~todd/cs344m/`
[3] `http://sourceforge.net/projects/rcroboviz/`
[4] `https://github.com/jstoecker/roboviz`

environments; our team, RoboCanes, is using this for use with our NAO robots. We expect this branch to become the primary version as it is polished for use in RoboCup 2013. Adding support for heterogeneous robot types would also be useful for 3D simulation.

Acknowledgments. The authors would like to thank Klaus Dorer of magmaOffenburg[5] and Drew Noakes for contributing to the source code of RoboViz. Thanks to Andreas Seekircher for the images used in figure 3.

References

1. 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)
2. Bödecker, J., Dorer, K., Rollmann, M., Xu, Y., Xue, F.: SimSpark User's Manual (June 2008)
3. Java Bindings for OpenGL (JOGL), `http://www.jogamp.org`
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. Technical Report 36, TZI, Universität Bremen (August 2006)
6. Laue, T., Röfer, T.: Simrobot-development and applications. In: The Universe of RoboCup Simulators-Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008). LNCS (LNAI). Springer, Heidelberg, Citeseer (2008)
7. Michel, O.: Webots: Professional Mobile Robot Simulation. Journal of Advanced Robotics Systems 1(1), 39–42 (2004)
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. (eds.) RoboCup 2000. LNCS (LNAI), vol. 2019, pp. 29–40. Springer, Heidelberg (2001)
10. Stoecker, J., Visser, U.: RoboViz: Programmable Visualization for Simulated Soccer. In: Röfer, T., Mayer, N.M., Savage, J., Saranlı, U. (eds.) RoboCup 2011. LNCS, vol. 7416, pp. 282–293. Springer, Heidelberg (2012)
11. TinMan. C-Sharp framework for 3D simulation league, `http://code.google.com/p/tin-man/`

[5] `http://robocup.fh-offenburg.de/html/index.htm`