vIsage - A visualization and debugging framework for distributed system applications

Christian Lipski
Computer Graphics Lab,
TU Braunschweig, Germany
lipski@cg.cs.tu-bs.de

Kai Berger
Computer Graphics Lab,
TU Braunschweig, Germany
berger@cg.cs.tu-bs.de

Marcus Magnor
Computer Graphics Lab,
TU Braunschweig, Germany
magnor@cg.cs.tu-bs.de

ABSTRACT

We present a *V*isualization, *S*imulation, *A*nd *G*raphical debugging *E*nvironment (vIsage) for distributed systems. Time-varying spatial data as well as other information from different sources can be displayed and superimposed in a single view at run-time. The main contribution of our framework is that it is not just a tool for visualizing the data, but it is a graphical interface for a simulation environment. Real world data can be recorded, played back or even synthesized. This enables testing and debugging of single components of complex distributed systems. Being the missing link between development, simulation and testing, e.g., in robotics applications, it was designed to significantly increase the efficiency of the software development process.

Keywords: Visualization, Simulation, Software Development Process, Monitoring, Application

1 INTRODUCTION

In many of today's applications, the processing of time-varying spatial data is distributed among various computers and architectures. This is the case for many scenarios such as scientific computing, massive multi-player online gaming or mobile robotics. With increasing complexity, the need to visualize the whole system's state as well as the need to debug single components in an isolated environment becomes evident.

We present a visualization, debugging and simulation environment that has been used in the context of the CarOLO project, where an autonomous vehicle has been developed. The vehicle participated in the finals of the Urban Challenge 2007. Initially designed as a stand-alone visualization client, our software is also capable of recording and synthesizing data, so that it can be used as a visual debugging tool as well. The paper is organized as follows: After giving an overview of related work in the area of visual debugging and monitoring in Section 2, the core idea of vIsage as a visualization client for distributed systems is presented in Section 3. Afterwards, the extensions to a simulation and debugging environment are explained in Section 4. The technical aspects of the overall software are regarded in detail in Section 5. The impact of the visual debugging tool on the software development process is discussed in Section 6 before we conclude our work in Section 7.

2 RELATED WORK

Our visual debugging system is related to the following previous work.

Visualizations of algorithms for debugging purposes extend the functionality of traditional text-based debuggers by showing data in processes in a graphical manner. The *Balsa* visualization tool by Brown and Sedgewick [BS84] names several participants in the visual debug process: the algorithm designer, the animator and the user. The *Zeus*-framework [BCA91] introduces a system which allows different views of the same data structure and the operating algorithms. The *Tango* Framework [Sta90] defines important parts of a debugging visualization, which are the image, its location, its path and a transition.
These approaches concentrate on debugging single processes and operate on a very elementary level. From a conceptual point of view, they are quite close to classical source code debuggers. In contrast to that, vIsage fulfills the need to visualize and debug complex distributed systems during run-time.

Visualizations of concurrent systems take into account that various objects coexist in different threads or systems. Although they are able to observe the internal states of these objects, their main focus lies on the information exchange among them. Jacobs et al.[JM03] applied abstraction techniques for UML diagrams are presented that try to reduce an object oriented system to its essential parts. Another object oriented debugging approach is proposed by Laffra et al. [LM94]. It displays instances of classes as an arrangement of animated squares. Method invocations are made visible by a change of color. The approach proposed by Vion-Dury et al. [VDS94] maps object instances to various geometric bodies and arranges them in a unique fashion. The visualization and debugging of distributed multi-agent systems are presented by Ndumu et al.[NNLC99]. Different tools are used

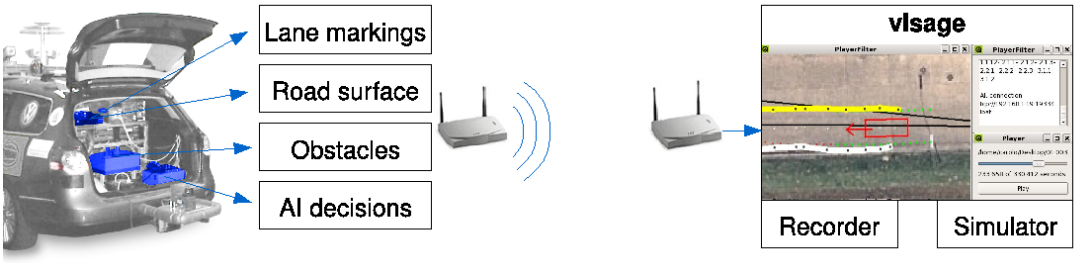


Figure 1: Components of a distributed system: Our application scenario of vIsage is autonomous driving. Several computers are controlling a vehicle. A vIsage client connects to the distributed system and visualizes the data. Data may be recorded or created synthetically using a simulator and support the development and debugging process.

to create different views on the data, so that users can concentrate on inspecting message interchange, agent tasks, internal states of the agents and statistics. Multi-agent systems may also be explored by exploiting the third dimension to visualize time [ISMT07]. Furthermore, a trade-off between completeness and clearness of presentation has to be considered when single agents are hidden from the user.
These approaches succeed in visualizing complex distributed systems, however, they do not offer any possibility to simulate interaction among them. vIsage enables the developers to work within a simulated environment without the need to access the actual working system.

Virtual or augmented reality tools are in use for debugging in the field of robotics. As robotics software processes primarily spatial data, it may be argued that this kind of software is designated for visual debugging. A selection of robotic development tools is presented by Tick[Tic06], the most recent one being the Microsoft Robotics Studio [CS07] which focuses on an easily accessible development interface and an integrated simulation environment. Robotic development tools are also used in the RoboCup competition, one of them is described by Penedo et al.[PPNC03]. Collet et al.[CM06] describe a shared perceptual space between an autonomous system and humans, e.g. developers. In this space an augmented reality is established, e.g. by enriching the video stream of an input camera with projected sensor data from the system. Obstacles can also be augmented into a camera image to assist human drivers, [TLWK07] uses laser scanners to detect objects in front of the car. In automotive design the idea of a virtual dashboard has been examined [BDGP+04], where several monitorable values, e.g. velocity, tire pressure and engine temperature, are displayed on an LCD-screen mounted at the conventional dashboard of a car. In the case of an emergency, e.g. when a distance sensor detects a possible impact, a warning message becomes visible on the virtual dashboard.
The augmented and virtual reality tools mentioned above can be seen as quite sophisticated and advanced, because they offer the possibility to visualize complex data in a virtual environment and may even simulate processes. However, they lack the ability to combine visualization, debugging, testing and simulation efforts into one coherent workflow. vIsage is a part of a toolchain that realizes this development paradigm.

Other aspects of the software development process and the algorithms used in the CarOLO project can be found in [BBR07], [LSB+08] and [BLL+08].

3 MONITORING DISTRIBUTED SYSTEMS

In automotive computer systems the need for a central monitoring and visualization tool has emerged in order to keep pace with the amount of data transferred among distributed components. By regarding only a single process in the system at a time, one will in most cases fail to detect the complex cause of erroneous behavior. In order to monitor the whole system, all relevant data that represent the communication among the single components and their internal processes have to be observed. vIsage offers the possibility to visualize all network data packages that are exchanged as well as arbitrary information about the internal state of single components. As all systems are synchronized in time via *ntp*, the system is able to put all incoming data packages into a well-defined temporal order. Out of vIsage's various possibilities to visualize data, the most frequently used ones are as follows.

Birds-eye view with external data. All data with a geometrical context can be displayed and superimposed in one bird's-eye view, Fig. 4. In our application, this includes position, orientation and velocity of the car, different sensor data and meta-data about the predefined road graph. Additional input, such as aerial or satellite photographs which match the car's

GPS position can be displayed as a background image. *WGS84* encoded data such as the car's road network graph are projected into a local Cartesian coordinate system. As the test areas were limited to a few square kilometers, projection errors have been negligible.

Log message view. System log messages of any computer can be observed with vIsage. As text messages are the most basic possibility to compile debugging data, the log message view is frequently used in early development stages. A global glimpse of the current system processes can easily be deducted by relating each computer's log messages.

Camera image view. Downsampled input images of currently connected cameras are shown in different widgets. While, developers can quickly assess the quality of the camera output, the general user can get a better understanding of the current situation.

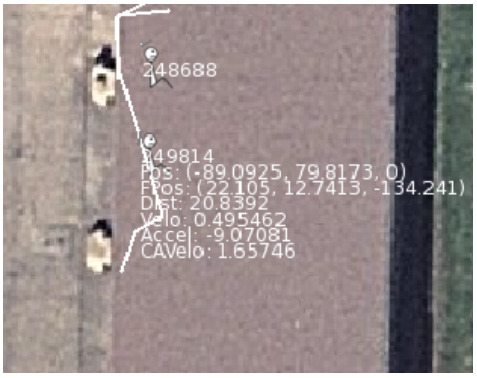
CAN bus message view. Similar to the computer system log messages additional communication protocols such as the car's CAN-bus can be monitored in a text window view with vIsage.

Graphical representation of artificial intelligence voting process. Internal processes within the artificial intelligence system, such as the evaluation of different steering wheel rotations or the determination of the current speed, are represented graphically.

In order to satisfy all developers' needs, each data source can be displayed in several media. E.g., for developing the actorics software, precise numerical values for velocity and throttle are required. For developers of other systems, a simplified graphical representation, such as an arrow of variable length or color are more beneficial.
The visualization of different data can interactively be turned on or off by the user. For some complex objects, additional information, such as acceleration and distance to other objects, can be displayed by clicking on their graphical representation, Fig.2. Since the system is able to access more than a dozen data sources that can be directed to the bird's-eye view, a different selection of displayed information is necessary for each individual developer. Furthermore, developers can define and save these visualization of the data they want to inspect at run-time.

Watchdog functionality. Due to the probability of faults in single hardware and software components, a watchdog system observes the status of all vital computers and processes. Local watchdog applications monitor all vital processes on every computer and their connected devices which have to send a so-called *heartbeat* in regular intervals. A dedicated watchdog computer keeps contact to the local instances and detects application crashes, application freezes and the overall availability of each computer. The watchdog's status evaluation is graphically accessible via vIsage, it even provides the possibility to manually restart the system or its components. Other meta information such as cycle times of certain processes and CPU load can also be monitored.

(a) original view (*magnified*)



(b) additional information (*magnified*)

Figure 2: By clicking on certain objects, additional information is revealed

4 VISUAL DEBUGGING

So far, means for monitoring and visualization have been presented. For later development stages, more functionality is required for an efficient workflow, Fig.6. The visualization engine is extended as follows.

Monitoring and visualizing, as described in chapter 3, are the core functionalities of vIsage and help the developers to identify erroneous or odd behavior. However, the reasons for failure may not be apparent at first sight. As the access to the working system may be limited or expensive, solving the problem in the

(a) Car (*magnified*)

(b) Road Graph (*magnified*)

(c) Lane (*magnified*)

(d) Obstacle (*magnified*)

Figure 3: Different entities are represented through different visualization layers: e.g. a car (red rectangle in a), a road network with traffic priorities (b), lane markings (c) and obstacles (d)

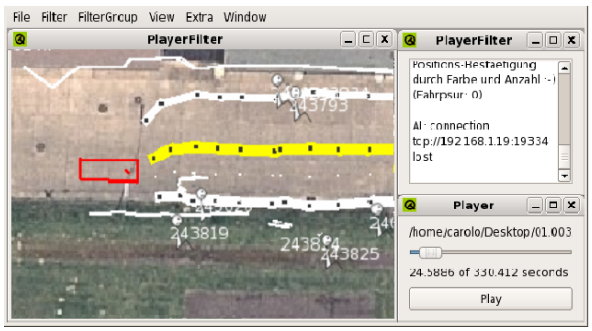
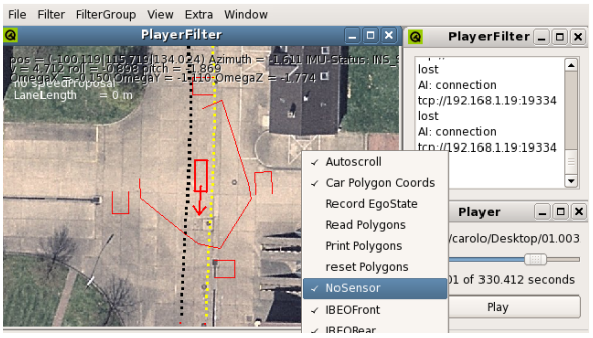


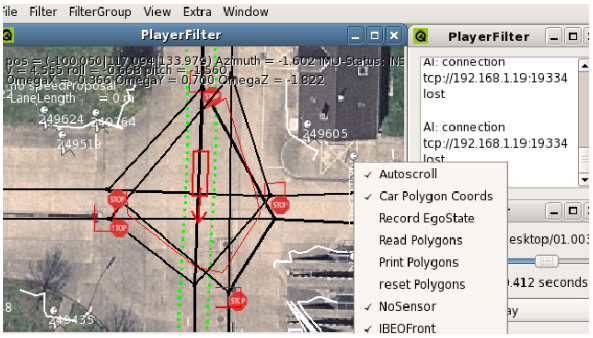
Figure 4: The vehicle's sensor data visualized in a bird's-eye view with satellite data. The red box represents the vehicle, the dotted lines recognized lane markings (cf. to Fig. 3)

lab can be very helpful. vIsage has been designed to be used as an interface for a virtual testing environment.

Recording all the data that are visualized with vIsage is also possible. Dedicated testers collect these data in case of odd or unexplainable behavior and analyze the data offline to find an explanation of the situation. If a problem remains unresolved, the data recordings are handed to the responsible developers, along with a description of the problem.



(a) Reduced Complexity



(b) Full Complexity

Figure 5: The information complexity in the top view can be interactively adjusted by disabling (a) or enabling (b) particular sensor data.

Playback of the recorded data to the isolated components, e.g., the artificial intelligence computer, may enable the developers to debug their software more efficiently. In combination with classical debugging tools, they use the recorded data as input to their system and try to discover the reasons for the behavior of their system. To ensure that the problem was solved entirely, the developers convert the actual data to a synthetic what-if scenario.

What-if scenarios can be created with vIsage and passed to the simulation framework. vIsage can be used to synthesize spatial data such as static obstacles and manually control dynamic objects, such as driving vehicles, during run-time. The developers create scenarios similar to the one recorded and provide for a robust handling of the given situation. In contrast to recorded data, these scenarios create the input data for the system dynamically so that alternative decision paths of the artificial intelligence can be examined. The resulting behavior of the simulation is visualized with vIsage.

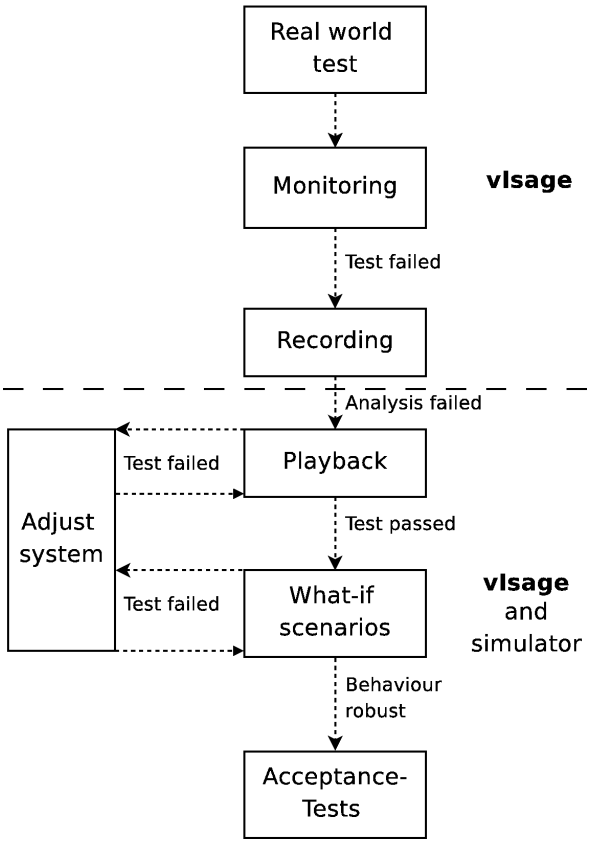
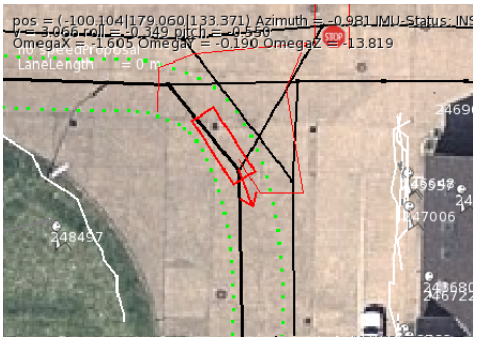
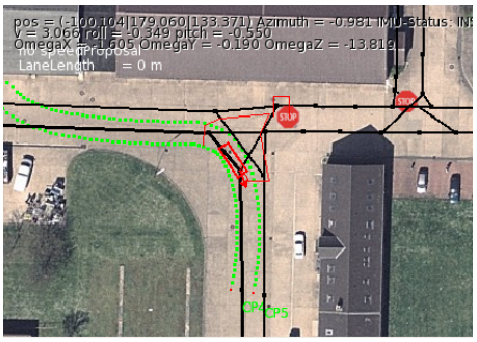


Figure 6: *vIsage* is the key link in the software development process. During real world tests monitoring and recording of the (odd) system behaviour are performed within vIsage. For bugfixing and system adjustment developers use the offline playback function. Finally complex what-if scenarious are created with vIsage in order to simulate them.



(a) Zoom in



(b) Zoom out

Figure 7: The scale of the view can be adjusted in order to get more detailed information (a) or to get the big picture (b).

Automatic Acceptance Tests are created and executed by the simulator to ensure that the system will still cope with these situations during later development stages. If necessary, the automatic simulation can be inspected with vIsage.

It is important to note that the usage of vIsage does not replace conventional source code debuggers. It establishes a very efficient and problem-oriented workflow that does not depend on or interfere with any development environment.

5 SYSTEM OVERVIEW

The main reasons for the system concept, Fig.8 are the requirement to access the data through a TCP connection and to allow the user to define which data are visualized in which view. To achieve these goals the communication is handled through a local data source which broadcasts on demand. An arbitrary number of vIsage clients receives the data from *data sinks*. A *data sink* is a process running on every computer, it is accessed by the individual applications, e.g., the sensor, artificial intelligence or actorics applications. Only the requested data is sent over the TCP connection. This enables a basic visualization on clients which are not connected via a broadband connection. For each different type of data there exists an individual *filter* which transforms the data into displayable objects. These can be either texts or geometric primitives which are displayed in the assigned *views*.

The user has the opportunity to adjust the display of information. He may scroll or zoom the bird's-eye view 7, data sources can be hidden or revealed via context menus 5.

vIsage was implemented on a Debian Linux system using the QT framework, but it can be easily ported to other operating systems. Even on low-end laptops, the vIsage system achieves real-time framerates.

6 RESULTS

vIsage was designed as a visual debugging and simulation tool for arbitrary distributed systems. As it was used in the CarOLO project, whose main goal was to

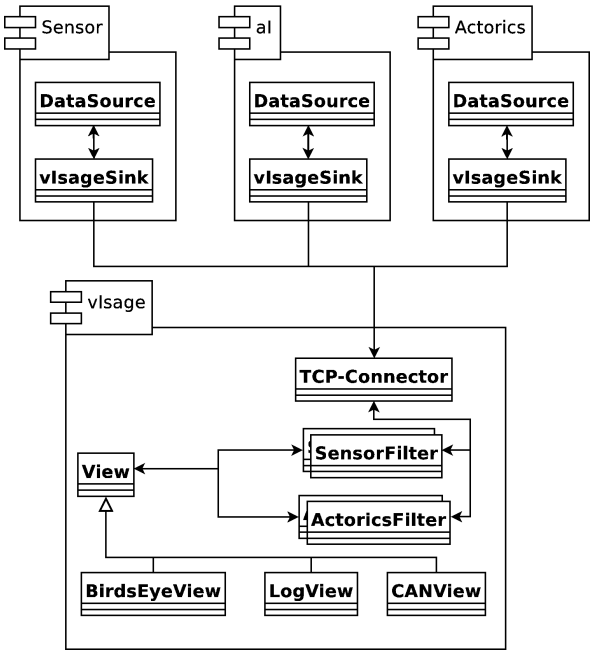


Figure 8: A simplified depiction of the architecture. Data sources stream their output to a data sink class, which is accessed by separate filters. The filters transform the received data into visualizable primitives which are then sent to their adjacent views.

develop an autonomous car, results from this particular software development process are given to show how the different functionalities of vIsage are integrated in the development toolchain. The developers of the vehicle's software system implemented the data structures that were exchanged over the network. They used vIsage as a front-end for visualization and development in the simulation environment. The testing team used vIsage to monitor vehicle test runs and to record single sessions. The vIsage developers implemented filters and views for all data objects and provided means to create what-if scenarios for the simulation environment.

The first impact of vIsage on the development process was observed in winter 2006/2007 during the early stages of the CarOLO project. During first tests with rather unstable software systems and hardware configurations, monitoring of distributed processes and computers was essential for debugging. Even when the software became more mature, hardware crashes occurred, e.g. due to harsh weather conditions. While testing on an abandoned military base in Germany during wintertime, several PCs crashed due to the low temperatures. Similar occurrences were observed on another testing site in Texas, when the air conditioning of the vehicle failed to sufficiently cool down the computers. In both cases, vIsage was vital for monitoring the heartbeats of the system and to tell apart system crashes from more complex error sources.

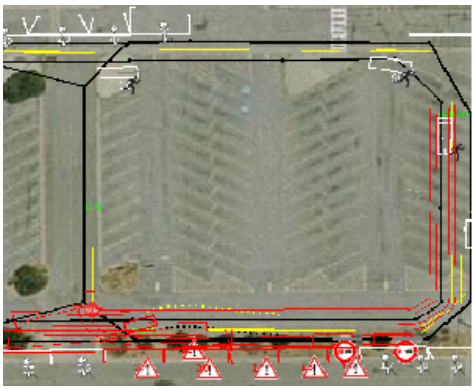


Figure 9: A qualification test of the DARPA Urban Challenge was recreated with vIsage, other dynamic vehicles (white rectangles) were added. The concrete barriers (red and white) were recreated using their original positions. The red triangles indicate that they were identified as obstacles on our vehicle's driving lane.

Recording of data became vital when multiple software systems worked stable and new functionality was developed in short intervals. The time assigned to each developer to operate on the actual vehicle was brief and many bugs had to be found off-line. E.g., it could be observed that the vehicle changed its driving lane multiple times without any obvious reason. Both systems involved, i.e. the lane detection system and the artificial intelligence, did not yield any signs of wrong behavior when being debugged in the lab. When the testing team recorded a session showing the erroneous lane changes, an analysis using vIsage revealed that the error occurred due to a receiver's misinterpretation of the so-called lane shift flag, that was exchanged between these two systems. The artificial intelligence interpreted a lane change to the left as a change to the right and tried to get back to its original lane.

In the final project phase during fall 2007, the use of the simulation environment in combination with vIsage became vital. The last and most important software change in the project was made possible by vIsage, when the vehicle entered the National Qualification Event of the DARPA Urban Challenge 2007. On one test course the vehicle kept changing from the outer to the inner lane, drove into oncoming traffic and changed to the reverse driving gear unexpectedly. This situation was recreated using vIsage and the erroneous behavior was reproduced. The artificial intelligence developers discovered a combination of unfortunate circumstances. The concrete borders of that course were too close to the outer lanes, so that the artificial intelligence interpreted them as static obstacles. In addition, the lanes were quite narrow and the high traffic density diminished the vehicle's possible paths too much. Several parameter alterations were made so that the vehicle managed to cope with this and similar situations. An

automatic acceptance test was tailored to this particular situation. Together with previous acceptance tests it ensured that this particular and other known situations could be handled with the most recent parameter settings.

7 CONCLUSION AND FUTURE WORK

vIsage is a valuable tool for the visualization and debugging of distributed systems that process spatial data. It ensured the success of the CarOLO project, where an autonomous vehicle was developed that participated in the finals of the DARPA Urban Challenge. The CarOLO project showed that complex systems require more than a plain visualization tool. With the new technical features, e.g. monitoring distributed heartbeats of the system parts and the record and replay of monitored data, *vIsage* differs from state-of-the-art visualization frameworks and is able to support the software developer in every stage of the software lifecycle.

A lot of possibilities exist when designing and extending an application such as vIsage. One question that arose during the design phase was about the use of three dimensional visualization modes. We decided against it, because most data can be more easily read and layered in two dimensions. By extending our system to visualize three dimensional data, objects like height fields from a laser scanner can be represented in a more intuitive manner. In addition, different views of the data can be realized and augmented with data such as the camera images.

One benefit of vIsage was the display of aerial photographs composed with the live data. This enables the use of vIsage as a tool for public demonstrations, as many people are familiar with these kinds of images from services like *Google maps*.

A very promising approach will be to extend the system so that more sensor inputs can be *simulated* in a realistic way. However, a complete simulation, including data simulating laser scanners and cameras, would enable a basic development of such systems without the need of an actual hardware setup. The time and ressource consuming test runs on the concrete hardware, e.g. on a vehicle, can start after a mature software revision is reached and different sensor setups have been evaluated.

Although it was mainly used for supporting the software development process in the CarOLO project, *vIsage* however is applicable in any geospatially related scenario, e.g. monitoring and debugging robot systems at the *RoboCup*, or integrating sensor network data, e.g. from distributed embedded devices like *Smart-Its* [BG03], into a global data map.

REFERENCES

[BBR07] C. Basarke, C. Berger, and B. Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication*, 4:1158–1174, 2007.

[BCA91] M.H. Brown, D.E.C.S.R. Center, and P. Alto. Zeus: a System for Algorithm Animation and Multi-View Editing. *Visual Languages*, pages 4–9, 1991.

[BDGP+04] F. Bellotti, A. De Gloria, A. Poggi, L. Andreone, S. Damiani, and P. Knoll. Designing Configurable Automotive Dashboards on Liquid Crystal Displays. *Cognition, Technology & Work*, 6(4):247–265, 2004.

[BG03] M. Beigl and H. Gellersen. Smart-Its: An Embedded Platform for Smart Objects. *Smart Objects Conference*, 2003, 2003.

[BLL+08] K. Berger, C. Lipski, C. Linz, T. Stich, and M. Magnor. The area processing unit of caroline - finding the way through darpa's urban challenge. *RobVis*, pages 260–274, February 2008.

[BS84] M.H. Brown and R. Sedgewick. A system for algorithm animation. *SIGGRAPH*, 18(3):177–186, 1984.

[CM06] T.H.J. Collett and B.A. MacDonald. Developer Oriented Visualisation of a Robot Program. *SIGCHI*, pages 49–56, 2006.

[CS07] E. Courses and T. Surveys. Microsoft Robotics Studio: a Technical Introduction. *Robotics & Automation Magazine, IEEE*, 14(4):82–87, 2007.

[ISMT07] S. Ilarri, J.L. Serrano, E. Mena, and R. Trillo. 3D Monitoring of Distributed Multiagent Systems. *WEBIST 07*, pages 978–972, 2007.

[JM03] T. Jacobs and B. Musial. Interactive Visual Debugging with UML. *Software visualization*, pages 115–122, 2003.

[LM94] C. Laffra and A. Malhotra. HotWire: a visual debugger for C++. *Proceedings of the 6th conference on USENIX Sixth C++ Technical Conference-Volume 6 table of contents*, pages 7–7, 1994.

[LSB+08] C. Lipski, B. Scholz, K. Berger, C. Linz, T. Stich, and M. Magnor. A fast and robust approach to lane marking detection and lane tracking. *SSIAI*, page to appear, March 2008.

[NNLC99] D.T. Ndumu, H.S. Nwana, L.C. Lee, and J.C. Collis. Visualising and Debugging Distributed Multi-Agent Systems. *Autonomous Agents*, pages 326–333, 1999.

[PPNC03] C. Penedo, J. Pavao, P. Nunes, and L. Custodio. Robocup Advanced 3D Monitor. *Proc. of RoboCup Symposium*, 2003.

[Sta90] J.T. Stasko. Tango: a Framework and System for Algorithm Animation. *Computer*, 23(9):27–39, 1990.

[Tic06] J. Tick. Convergence of Programming Development Tools for Autonomous Mobile Research Robots. *SISY*, pages 29–30, 2006.

[TLWK07] M. Tonnis, R. Lindl, L. Walchshausl, and G. Klinker. Visualization of Spatial Sensor Data in the Context of Automotive Environment Perception Systems. *ISMAR 07*, 2007.

[VDS94] J.Y. Vion-Dury and M. Santana. Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems. *Object-oriented programming systems, language, and applications*, pages 65–84, 1994.