SSL-Vision: The Shared Vision System for the RoboCup Small Size League

Stefan Zickler[1], Tim Laue[2], Oliver Birbach[2], Mahisorn Wongphati[3], and Manuela Veloso[1]

[1] Carnegie Mellon University, Computer Science Department,
5000 Forbes Ave., Pittsburgh, PA, 15213, USA
E-Mail: {szickler, veloso}@cs.cmu.edu
[2] Deutsches Forschungszentrum für Künstliche Intelligenz GmbH,
Safe and Secure Cognitive Systems, Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
E-Mail: {Tim.Laue, Oliver.Birbach}@dfki.de
[3] Chulalongkorn University,
254 Phyathai Road, Patumwan, Bangkok, 10330, Thailand
E-Mail: mahisorn.w@gmail.com

Abstract. The current RoboCup Small Size League rules allow every team to set up their own global vision system as a primary sensor. This option, which is used by all participating teams, bears several organizational limitations and thus impairs the league's progress. Additionally, most teams have converged on very similar solutions, and have produced only few significant research results to this global vision problem over the last years. Hence the responsible committees decided to migrate to a shared vision system (including also sharing the vision hardware) for all teams by 2010. This system – named SSL-Vision – is currently developed by volunteers from participating teams. In this paper, we describe the current state of SSL-Vision, i. e. its software architecture as well as the approaches used for image processing and camera calibration, together with the intended process for its introduction and its use beyond the scope of the Small Size League.

1 Introduction

Given the current rules of the RoboCup Small Size League (SSL) [1], every team is allowed to mount cameras above or next to the field. There has also been an option of using local instead of global vision, but this turned out to be not competitive. For adequately covering the current field, most teams prefer to use two cameras, one above each half. This configuration bears one major problem (implicating a set of sub-problems): The need for long setup times before as well as during the competition. Having five teams playing on a field, ten cameras need to be mounted and calibrated. During these preparations, a field cannot be used for any matches or other preparations. Due to this situation, teams are always bound to their field (during one phase of the tournament) and unable to play any testing matches against teams from other fields. Hence the Small Size

League needs as many fields as it has round robin groups. Flexible schedules as in the Humanoid or the Standard Platform League – which have more teams but need less fields – are currently impossible.

In the future, these problems might become even worse since the current camera equipment already reached its limits. Having a larger field – which is a probable change for 2010, given the common field with the Humanoid and the Standard Platform League –, every team will be forced to set up four cameras above the field. This would significantly increase preparation times during a competition and decrease time and flexibility for scheduling matches.

To overcome this situation, the SSL committees decided to migrate to a shared vision system, i. e. to a single set of cameras per field which are connected to an image processing server which broadcasts the vision output to the participating teams. The software for this server needs to be flexible, i. e. scalable for future changes and open to new approaches, as well as competitive, i. e. performant and precise, to not constrain the current performance of the top teams. This system, named SSL-Vision, is now developed by a group of volunteers from the SSL. This paper describes the current state and the future of this project.

The paper is organized as follows: Section 2 describes the overall architecture of the system. The current approaches for image processing and camera calibration are presented in Section 3. The paper concludes with a description of the system's introduction and the resulting implications in Section 4.

2 Framework

SSL-Vision is intended to be used by all Small Size League teams, with a variety of camera configurations and processing hardware. As such, configurability and robustness are key design goals for its framework architecture. Additionally, the project's collaborative openness and its emphasis on research all need to be reflected in its framework architecture through an extendable, manageable, and scalable design.

One major design goal for the framework is to support concurrent image processing of multiple cameras in a single seamless application. Furthermore, the application should integrate all necessary vision functionality, such as configuration, visualization, and actual processing. To achieve better scalability on modern multi-core and hyper-threaded architectures, the application uses a multi-threaded approach. The application's main thread is responsible for the Graphical User Interface (GUI), including all visualizations, and configuration dialogs. Additionally, each individual camera's vision processing is implemented in a separate thread, thus allowing truly parallel multi-camera capture and processing. The application is implemented in C++ and makes heavy use of the Qt toolkit [2], to allow for efficient, platform-independent development.

Fig. 1 shows an overview of the framework architecture. The entire system's desired processing flow is encoded in a *multi-camera stack* which fully defines how many cameras are used for capturing, and what particular processing should be performed. The system has been designed so that developers can create different

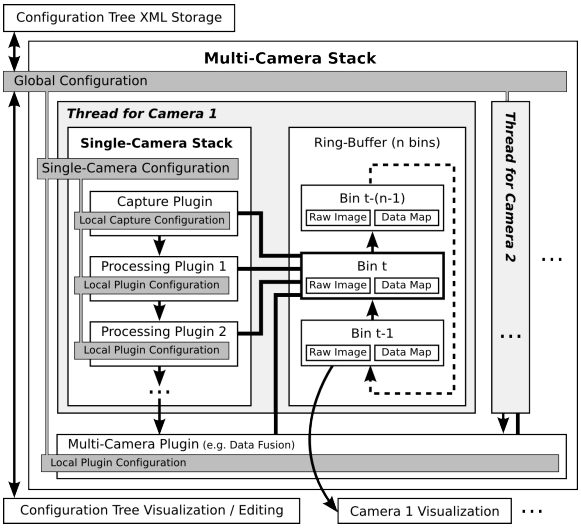


Fig. 1. The extendible, multi-threaded processing architecture of SSL-Vision.

stacks for different robotics application scenarios. By default, the system will load a particular multi-camera stack, labeled the "RoboCup Small Size Dual Camera Stack" which we will elaborate on in the following section. However, the key point is that the SSL-Vision framework provides support for choosing any arbitrarily complex, user-defined stack at start-up, and as such becomes very extendible and even attractive for applications that go beyond robot soccer.

Internally, a multi-camera stack consists of several threads, each representing the processing flow of a corresponding capture device. Each thread's capturing and processing flow is modeled as a *single-camera stack*, consisting of multiple *plugins* which are executed in order. The first plugin in any single-camera stack implements the image capturing task. All capture plugins implement the same C++ capture interface, thus allowing true interchangeability and extendibility of capture methods. The framework furthermore supports unique, independent configuration of each single-camera stack, therefore enabling capture in heterogeneous multi-camera setups. Currently, the system features a capture plugin supporting IEEE 1394 / DCAM cameras, including higher bandwidth Firewire 800 / 1394B ones. Configuration and visualization of all standard DCAM parameters (such as white balance, exposure, or shutter speed) is provided through the GUI at run-time, thus eliminating the need for third-party DCAM parameter configuration tools. The system furthermore features another capture plugin supporting capturing from still image and video files, allowing development on machines which do not have actual capture hardware. Additional capture plugins for Gigabit Ethernet (GigE) Vision as well as Video4Linux are under construction as well.

2.1 The Capture Loop

A capture plugin produces an output image at some resolution, in some color-space. For further processing, this image data is stored in a *ring-buffer* which is internally organized as a cyclical linked-list where each item represents a *bin*, as is depicted in Fig. 1. On each capture iteration, the single-camera stack is assigned a bin where it will store the captured image and any additional data resulting from processing this image. As the stack is being executed, each of its plugins is sequentially called, and each of them is able to have full read and write access to the data available in the current bin. Each bin contains a *data map*, which is a hash-map that is able to store arbitrary data under a meaningful label. This data map allows a plugin to "publish" its processing results, thus making them available to be read by any of the succeeding plugins in the stack.

The purpose of the ring-buffer is to allow the application's visualization thread to access the finished processing results while the capture thread is allowed to already move on to the next bin, in order to work on the latest video frame. This architecture has the great advantage of not artificially delaying any image processing for the purpose of visualization. Furthermore, this ring-buffered, multi-threaded approach makes it possible to prioritize the execution schedule of the capture threads over the GUI thread, thus minimizing the impact of visualization on processing latency. Of course it is also possible to completely disable all visualizations in the GUI for maximum processing performance.

In some processing scenarios it is necessary to synchronize the processing results of multiple camera threads after all the single-stack plugins have finished executing. This is done through optional *multi-camera plugins*. A typical example would be a plugin which performs the data fusion of all the threads' object detection results and then sends the fused data out to a network.

2.2 Parameter Configuration

Configurability and ease of use are both important goals of the SSL-Vision framework. To achieve this, all configuration parameters of the system are represented in a unified way through a variable management system called *VarTypes* [3]. The VarTypes system allows the organization of parameters of arbitrarily complex types while providing thread-safe read/write access, hierarchical organization, real-time introspection/editing, and XML-based data storage.

Fig. 1 shows the hierarchical nature of the system's configuration. Each plugin in the SSL-Vision framework is able to carry its own set of configuration parameters. Each single-camera stack unifies these local configurations and may additionally contain some stack-wide configuration parameters. Finally, the multi-camera stack unifies all single-camera stack configurations and furthermore contains all global configuration settings. This entire configuration tree can then be seamlessly stored as XML. More importantly, it is displayed as a data-tree during runtime and allows real-time editing of the data. Fig. 2 shows a snapshot of the data-tree's visualization.

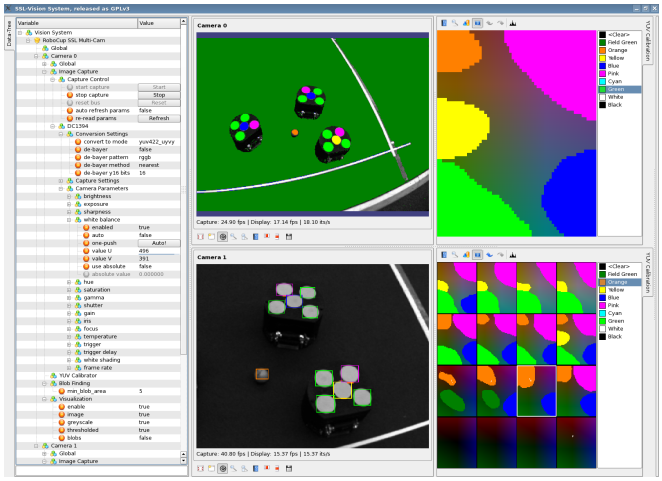


Fig. 2. Screenshot of SSL-Vision, showing the parameter configuration tree (left), live-visualizations of the two cameras (center), and views of their respective color thresholding YUV LUTs (right).

3 RoboCup SSL Image Processing Stack

The system's default multi-camera stack implements a processing flow for solving the vision task encountered in the RoboCup Small Size League. In the Small Size League, teams typically choose a dual-camera overhead vision setup. The robots on the playing field are uniquely identifiable and locatable based on colored markers. Each robot carries a team-identifying marker in the center as well as a unique arrangement of additional colored markers in order to provide the robot's unique ID and orientation. In the past, each team was able to determine their own arrangement and selection of these additional markers. However, with the introduction of the SSL-Vision system, it is planned to unify the marker layout among all teams for simplification purposes.

The processing stack for this Small Size League domain follows a typical multi-stage approach as it has been proven successful by several teams in the past. The particular single-camera stack consists of the following plugins which we explain in detail in the forthcoming sections: image capture, color thresholding, runlength encoding, region extraction and sorting, conversion from pixel coordinates to real-world coordinates, pattern detection and filtering, and delivery of detection results via network.

3.1 CMVision-Based Color Segmentation

The color segmentation plugins of this stack, namely color thresholding, runlength-encoding, region extraction and region sorting, have all been implemented by porting the core algorithms of the existing CMVision library to

the new SSL-Vision plugin architecture [4]. To perform the color thresholding, CMVision assumes the existence of a lookup table (LUT) which maps from the input image's 3D color space (by default YUV), to a unique color label which is able to represent any of the marker colors, the ball color, as well as any other desired colors. The color thresholding algorithm then sequentially iterates through all the pixels of the image and uses this LUT to convert each pixel from its original color space to its corresponding color label. To ease the calibration of this LUT, the SSL-Vision system features a fully integrated GUI which is able to not only visualize the 3D LUT through various views, but which also allows to directly pick calibration measurements and histograms from the incoming video stream. Fig. 2 shows two example renderings of this LUT. After thresholding the image, the next plugin performs a line-by-line runlength encoding on the thresholded image which is used to speed up the region extraction process. The region extraction plugin then uses CMVision's tree-based union find algorithm to traverse the runlength-encoded version of the image and efficiently merge neighboring runs of similar colors. The plugin then computes the bounding boxes and centroids of all merged regions and finally sorts them by color and size.

3.2 Camera Calibration

In order to deduce information about the objects on the field from the measurements of the cameras, a calibration defining the relationship between the field geometry and the image plane is needed. Depending on the applied calibration technique, current teams use a variety of different calibration patterns, leading to an additional logistic effort while attending tournaments. Furthermore, many such calibration procedures require the patterns to cover the field partially or as a whole, making the field unusable for other teams during the setup.

For the calibration procedure of SSL-Vision, no additional accessories are required. Instead, the procedure uses solely the image of the field and the dimensions defined in the league's rules. Because SSL-Vision uses two independent vision stacks, we calibrate both cameras independently using the corresponding half field. To model the projection into the image plane, a pin-hole camera model including radial distortion is used. The corresponding measurement function h projects a three-dimensional point M from the field into a two-dimensional point m in the image plane. The model parameters for this function are, intuitively, the orientation q and the position t, transforming points from coordinate system of the field into the coordinate system of the camera, and the intrinsic parameters f, (u_0, v_0) and κ indicating the focal-length, image center and radial distortion, respectively.

In a typical Small Size League camera setup, estimating such a set of calibration parameters by using only the field dimensions is actually ill-posed, due to the parallelism of the image plane and the field plane (which is the reason for the frequent use of calibration patterns). The estimator cannot distinguish whether the depth is caused by the camera's distance from the field (encoded in t_z) or the focal length (encoded in f). To circumvent this problem, a man-

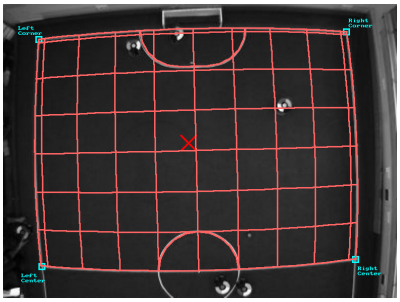
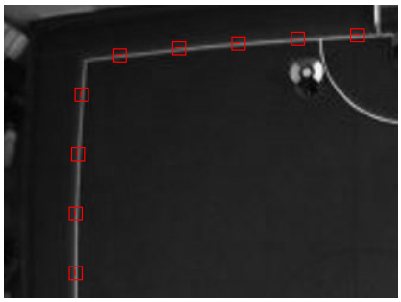
a)  b) 

Fig. 3. Camera calibration: a) Result of calibration. The points selected by the user are depicted by labeled boxes, the field lines and their parallels are projected from the field to the image plane. b) Detected edges for the second calibration step.

ual measurement of the camera's distance from the field is performed and the parameter is excluded from the estimation algorithm.

The actual calibration procedure consists of two steps:

1. The user selects the four corner points of the half-field in the camera image. Based on the fact that the setup is constrained by the rules, rough but adequate initial parameters can be determined in advance. Based on these initial parameters, a least squares optimization is performed to determine a set of parameters corresponding to the marked field points [5] (cf. Fig. 3a). Thus, we want to minimize the squared difference of the image points m_i that were marked by the user in the image plane and corresponding field point M_i, projected into the image plane using the measurement function mentioned above:

$$\sum_{i=1}^{4} |m_i - h(M_i, q, \boldsymbol{t}, f, u_o, v_o, \kappa)|^2 \tag{1}$$

   Since this is a nonlinear least squares problem, the Levenberg-Marquardt algorithm [6] is used to find the optimal set of parameters.
2. After this first estimate, the parameters are refined by integrating segments of field lines into the estimation. Since the field lines contrast with the rest of the field, an edge-detector is applied to find the lines using their predicted position computed from the estimate and the field dimensions as a search window (cf. Fig. 3b). A reasonable number of edges on the lines is then used to extend the least squares estimation. For this, we introduce a new to be estimated parameter α for each measurement and minimize the deviation of the measured point on the field line and the projection of the point $(\alpha\, p_1 + (1-\alpha)\, p_2$ between the two points p_1, p_2 constraining the line segment. The term to be minimized now reads

$$\sum_{i=1}^{4} |m_i - h(M_i, p)|^2 + \sum_{i=1}^{n} |m_i - h(\alpha_i\, L_{i,1} + (1-\alpha_i)\, L_{i,2}, p)|^2 \tag{2}$$

where $L_{i,1}$ and $L_{i,2}$ constrain the line segment and α_i describes the actual position of measurement i on this line. Please note, that multiple measurements may lie on the same line. For better readability, the camera parameters were combined into p.

After this calibration procedure, the inverted measurement function h^{-1} can be used to transform pixel coordinates to real-world coordinates.

3.3 Pattern Detection

After all regions have been extracted from the input image and all their real-world coordinates have been computed, the processing flow continues with the execution of the pattern recognition plugin. The purpose of this plugin is to extract the identities, locations, and orientations of all the robots, as well as the location of the ball. The internal pattern detection algorithm was adopted from the CMDragons vision system and is described in detail in a previous paper [7].

Although this pattern detection algorithm can be configured to detect patterns with arbitrary arrangements of 2D colored markers, the Small Size committees are intending to mandate a standard league-wide pattern layout with the transition to SSL-Vision, for simplification purposes.

3.4 System Integration and Performance

After the pattern detection plugin has finished executing, its results are delivered to participating teams via UDP Multicast. Data packets are encoded using Google Protocol Buffers [8], and contain positions, orientations, and confidences of all detected objects, as well as additional meta-data, such as a timestamp and frame-number. Furthermore, SSL-Vision is able to send geometry data (such as camera pose) to clients, if required. To simplify these data delivery tasks, SSL-Vision provides a minimalistic C++ sample client which teams can use to automatically receive and deserialize all the extracted positions and orientations of the robots and the ball. Currently, SSL-Vision does not perform any "sensor fusion", and instead will deliver the results from both cameras independently, leaving the fusion task to the individual teams. Similarly, SSL-Vision does not perform any motion tracking or smoothing. This is due to the fact that robot tracking typically assumes knowledge about the actual motion commands sent to the robots, and is therefore best left to the teams.

Table 1 shows a break-down of processing times required for a single frame of a progressive YUV422 video stream of 780×580 pixel resolution. These numbers represent rounded averages over 12 consecutive frames taken in a randomly configured RoboCup environment, and were obtained on an Athlon 64 X2 4800+ processor.

3.5 GPU-Accelerated Color Thresholding

The traditional sequential execution of CMVision's color thresholding process is – despite its fast implementation through a LUT – a very computationally

Table 1. Single frame processing times for the plugins of the default RoboCup stack.

Plugin	Time
Image capture	1.1 ms
Color thresholding (CPU)	3.6 ms
Runlength encoding	0.7 ms
Region extraction and sorting	0.2 ms
Coordinate conversion	< 0.1 ms
Pattern detection	< 0.1 ms
Other processing overhead	0.4 ms
Total frame processing	< 6.2 ms

Table 2. Single frame processing times for the naïve GPU-accelerated color thresholding.

Component	Time
Copy data to texture memory	3.0 ms
Color thresholding (GPU)	32 μs
Copy data from frame buffer	11.0 ms
Total thresholding time	< 15 ms

intensive process. The performance values in Table 1 clearly show that color thresholding currently constitutes the latency bottleneck of the processing stack by a significant margin. One of the best ways to overcome this latency problem is by exploiting the fact that color thresholding is a massively parallelizable problem because all pixels can be processed independently. However, even with the reasonable price and performance in current Personal Computers, only 2 or 4 physical CPU cores are available for parallel computing which in our case are already occupied by each camera's capture threads, the visualization process, and other OS tasks. Nevertheless, modern commodity video cards which feature a programmable Graphic Processing Unit (GPU) have become widely available in recent years. Because GPUs are inherently designed to perform massively parallel computations, they represent a promising approach for hardware-accelerated image processing. In this section we will provide initial evaluation results of a GPU-accelerated color thresholding algorithm which may be included in a future release of SSL-Vision.

To implement the GPU-accelerated thresholding algorithm, we selected the OpenGL Shading Language (GLSL), due to its wide support of modern graphics hardware and operating systems. GLSL allows the programming of the graphics hardware's *vertex processor* and *fragment processor* through the use of small programs known as *vertex shaders* and *fragment shaders*, respectively [9]. Vertex shaders are able to perform operations on a per-vertex basis, such as transformations, normalizations, and texture coordinate generation. Fragment shaders (also commonly referred to as *pixel shaders*), on the other hand, are able to perform per-pixel operations, such as texture interpolations and modifications.

Because we are interested in performing color thresholding on a 2D image, we implement our algorithm via a fragment shader. Fig. 4 shows an overview of this GPU-accelerated color thresholding approach. First, before any kind of video processing can happen, we need to define a thresholding LUT. This LUT is similar to the traditional CMVision version in that it will map a 3D color input (for example in RGB or YUV) to a singular, discrete color label. The difference is however, that this LUT now resides in video memory and is internally rep-

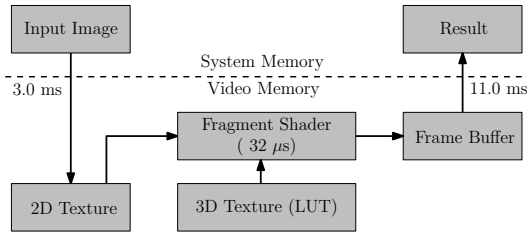


Fig. 4. Block diagram of color thresholding using GLSL

resented as a 3D texture which can be easily accessed by the fragment shader. As modern video hardware normally provides 128MB video memory or more, it is easily possible to encode a full resolution LUT (256x256x256, resulting in approximately 17MB). In order to perform the actual color thresholding processing, any incoming video frame first needs to be copied to the video hardware to be represented as a 2D texture that the shader will be able to operate on. The fragment shader's operation then is to simply replace a given pixel's 3D color value with its corresponding color label from the 3D LUT texture. We apply this shader by rendering the entire 2D input image to the frame buffer. After the render process, we now need to transfer the labeled image from the frame buffer back to system memory for further processing by any other traditional plugins.

Table 2 shows the average time used by each step of the color thresholding process using an NVIDIA Geforce 7800 GTX video card under Linux, on the same CPU that was used for the measurements in Table 1. The input video data again had a size of 780×580 pixels. The values clearly indicate that the actual thresholding step is about 100 times faster than on the normal CPU. Interestingly, however, this approach has introduced two new bottlenecks in the upload and download times between system memory and video memory which, in total, makes this approach more than four times slower than the traditional color thresholding routine.

A potential approach for improving this performance would be to convert most or all other image-processing related plugins, which follow color thresholding, to the GPU. This way, there would be no requirement to transfer an entire image back from video memory to system memory. Instead, a major portion of the image processing stack would be computed on the GPU, and only the resulting data structures, such as final centroid locations, could be transfered back to system memory. For this process to work however, the color segmentation algorithms would need to be majorly revised, and as such this approach should be considered future work.

4 Further Steps and Implications

Beyond a proper technical realization, as described in the previous sections, the introduction of a shared vision system for the Small Size League bears several organizational issues as well as implications for future research, even for other RoboCup leagues.

4.1 Schedule of Introduction

The first release of SSL-Vision has been published in spring 2009. Since then, all teams are free to test the application in their labs, to review the code, and to submit improvements. At the upcoming regional competitions as well as at RoboCup 2009, the usage of the system is voluntary, i. e. teams can run it on their own computers or decide to share the vision system with others. However, everybody is free to keep using their own system. After this transition phase, which has been established to provide a rehearsal under real competition conditions, the usage of SSL-Vision will become obligatory, in time for RoboCup 2010.

4.2 Implications for Research

By introducing a shared vision system for all teams, one degree of individuality for solving the global task "Playing Soccer with Small Size Robots" becomes removed. However, during the last years, most experienced teams have converged towards very similar sensing solutions, and have produced only few significant research results regarding computer vision. De facto, having a performant vision system does not provide any major advantage, but should rather be considered a minimum requirement as sophisticated tactics and precise control are dominant factors in the SSL. On the other hand, new teams often experience problems having an insufficient vision application which strongly decreases their entire system's performance. Thus, SSL-Vision will directly benefit all newcomers, allowing them to base their tactics on a robust global vision sensor.

Furthermore, the transition to a shared vision system does not imply a stagnation in vision-related research. In fact, due to its open and modular architecture (cf. Sect. 2), SSL-Vision allows researchers to develop and "plug in" novel image processing approaches without needing to struggle with technical details (e.g. camera interface control or synchronization). Therefore, new approaches can be fairly and directly compared with existing ones, thus ensuring a continuing, community-driven evolution of SSL-Vision's processing capabilities and performance.

Whereas the system's impact for the Small Size League is obvious, it might also become directly useful for teams in other RoboCup leagues. Many researchers in local vision robot leagues require precise reference data – so-called *ground truth* – to evaluate their results during development, e. g. of localization algorithms or for gait optimization. One example for tracking humanoid soccer robots with an SSL vision system is shown in [10]. Due to the standardized field

size, SSL-Vision becomes an off-the-shelf solution for the Humanoid as well as the Standard Platform League.

Finally, it needs to be strongly emphasized that SSL-Vision's architecture is not at all limited to only solving the task of robot soccer vision. Instead, the system should really be recognized as a framework which is flexible and versatile enough to be employed for almost any imaginable real-time image processing task. While, by default, the system provides the stacks and plugins aimed at the RoboCup domain, we are also eagerly anticipating the use and extension of this system for applications which go beyond robot soccer.

5 Conclusion

In this paper, we introduced the shared vision system for the RoboCup Small Size League, called SSL-Vision. We presented the system's open software architecture, described the current approaches for image processing and camera calibration, and touched upon possible future improvements, such as GPU-acceleration. Finally, we discussed SSL-Vision's scheduled introduction and its impact on research within the RoboCup community. We strongly believe that the system will positively affect the Small Size League by reducing organizational problems and by allowing teams to re-focus their research efforts towards elaborate multi-agent systems and control issues. Because SSL-Vision is a community project, everybody is invited to participate. Therefore, SSL-Vision's entire codebase is released as open-source [11].

References

1. RoboCup Small Size League: SSL Web Site. `http://small-size.informatik.uni-bremen.de` (2009)
2. Nokia Corporation: The Qt Toolkit. `http://www.qtsoftware.com/`
3. Zickler, S.: The VarTypes System. `http://code.google.com/p/vartypes/`
4. Bruce, J., Balch, T., Veloso, M.: Fast and inexpensive color image segmentation for interactive robots. In: Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '00). Volume 3. (2000) 2061–2066
5. Zhang, Z.: A flexible new technique for camera calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence **22**(11) (2000) 1330–1334
6. Marquardt, D.: An algorithm for least-squares estimation of nonlinear parameters. SIAM Journal on Applied Mathematics **11**(2) (1963) 431–441
7. Bruce, J., Veloso, M.: Fast and accurate vision-based pattern detection and identification. In: Proceedings of the 2003 IEEE International Conference on Robotics and Automation (ICRA '03). (2003)
8. Google Inc.: Protocol Buffers. `http://code.google.com/p/protobuf/`
9. Rost, R.J.: OpenGL Shading Language (2nd Edition). Addison-Wesley Professional (2006)
10. Laue, T., Röfer, T.: Particle filter-based state estimation in a competitive and uncertain environment. In: Proceedings of the 6th International Workshop on Embedded Systems, Vaasa, Finland. (2007)
11. SSL-Vision Developer Team: RoboCup Small Size League Shared Vision System Project Home. `http://code.google.com/p/ssl-vision/` (2009)