CVVisual: Interactive Visual Debugging of Computer Vision Programs

Andreas Bihlmaier
Institute for Anthropomatics and Robotics (IAR),
Intelligent Process Control and Robotics Lab (IPR),
Karlsruhe Institute of Technology (KIT),
76131 Karlsruhe, Germany
Email: andreas.bihlmaier@kit.edu

Heinz Wörn
Institute for Anthropomatics and Robotics (IAR),
Intelligent Process Control and Robotics Lab (IPR),
Karlsruhe Institute of Technology (KIT),
76131 Karlsruhe, Germany
Email: woern@kit.edu

***Abstract*—Most computer vision applications are built from a combination of basic computer vision algorithms, such as filters, descriptors and matchers. The functionality of this computer vision toolbox is well understood and solid implementations exist. One of the leading and most often used implementations is the Open Source Computer Vision Library (OpenCV), which implements more than 500 computer vision algorithms. However, most of these algorithms have multiple parameters that have to be tuned to the specific vision application. Therefore, during development of new vision applications, some human-in-the-loop iteration cannot be avoided. The problem is that OpenCV lacks support for this phase of development. This usually leads to a situation where each developer creates his own static ad-hoc debug solutions that output intermediate results in a printf-debugging manner. To remedy this situation, we developed reusable general-purpose interactive tools for visual debugging and development of OpenCV-based computer vision applications.**

I. Introduction

In the paper at hand we introduce neither new computer vision algorithms nor a new computer vision application. Rather, we focus at the intermediate stage between basic vision algorithms and working vision applications, i.e. we focus on the development phase of computer vision solutions. Provided that a toolbox of basic vision algorithms is available, there still remain a lot of problems to be solved in order to arrive at a working application. These problems are not only at the level of designing the image filter pipeline. Also during the implementation of the design many development iterations are required. In these iterations the application specific suitability of elements in the computer vision toolbox together with working parameter sets have to be identified. Due to the lack of tools to support this stage, which in our experience often takes up a majority of the total development time – at least in a research setting – each developer has to constantly (re-)invent ad-hoc solutions for it. This often leads to code fragments such as the one shown in Fig. 1. To improve this situation, we developed CVVisual as a module for the popular Open Source Computer Vision Library (OpenCV)[1] that provides interactive visual debugging and development tools for computer vision applications. Instead of constantly modifying the static ad-hoc solutions in a copy-and-paste programming manner, CVVisual provides the developer with generic, reusable and interactive API functions. With CVVisual the printf-debugging style code fragment from above (Fig. 1) becomes Fig. 2. This reduces the development effort and amount of conditionally compiled non-application code, thereby increasing developer productivity and code readability. In addition, as will be shown in the following, the framework behind the CVVisual calls provides much more functionality than the ad-hoc solutions.

```
cv::dilate(imgEdge, imgEdgeDilate, cv::Mat());
#ifdef DEBUG_VISUALIZATION
cv::namedWindow("edges");
cv::imshow("edges", imgEdge);
cv::namedWindow("dilate");
cv::imshow("dilate", imgEdgeDilate);
cv::Mat imgChanged; // very simple visualization
cv::compare(imgEdge, imgEdgeDilate,
            imgChanged, cv::CMP_NE);
cv::namedWindow("changed");
cv::imshow("changed", imgChanged);
#endif
```

Fig. 1. Example of common type of debug code found in OpenCV-based vision applications. When a specific visualization is required, the debug code can become as long and elaborate as the surrounding application code. This type of code is seldom reusable and thus often reused in a copy-and-paste manner. Furthermore, if multiple stages of the vision pipeline have to be analyzed either too many debug windows are opened or they are reused, thereby statically deciding on which information is lost. The same holds true when dealing with streaming data.

```
cv::dilate(imgEdge, imgEdgeDilate, cv::Mat());
cvv::debugFilter(imgEdge, imgEdgeDilate,
                 CVVISUAL_LOCATION, "filtered");
```

Fig. 2. The code example shows how the same problem as in Fig. 1 can be solved with CVVisual. Once the `cvv::debugFilter` call is executed its parameters are added to the CVVisual overview window (Fig. 3), from which they can be interactively inspected (Fig. 6).

We already validated CVVisual in an academic research context. For example, during the development of vision algorithms for an endoscope guidance robot in minimally-invasive surgery [1] [2]. The algorithms comprised real-time image stitching, segmentation and classification tasks. Although working with the framework (cf. Fig. 2) already provides several benefits compared to the previous method (cf. Fig. 1) of debugging OpenCV algorithms, additional coverage of other

[1] http://opencv.org/

978-1-4673-7929-8/15/$31 © 2015 IEEE

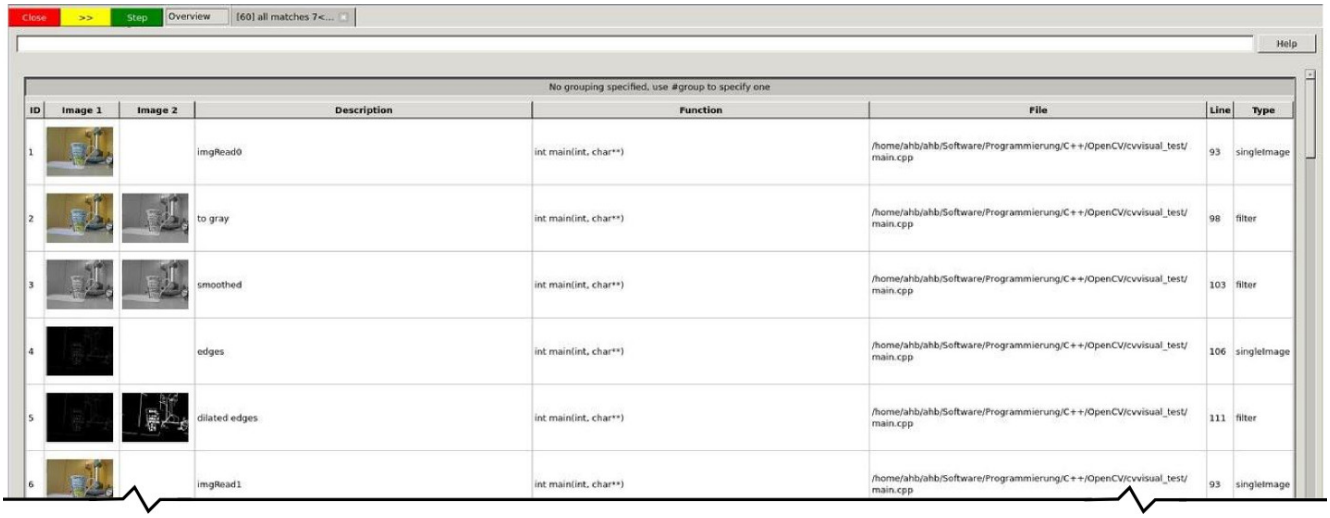

Fig. 3. The image shows the main overview window of CVVisual. All `cvv` API calls add one line to this overview. A powerful search, filter and sort engine is integrated to handle large amounts of image data. For each of the API calls, there exists one or multiple perspectives to interactively inspect the image data (see Fig. 5,6,7,8). These appear as dockable tabs of the main window. The three buttons in the top left corner allow to unblock the program until the next `cvv` call or to continue the program without blocking on `cvv` calls until `cvv::finalShow` is encountered (cf. Sect. III).

OpenCV modules would create further gains. CVVisual was published as open source and has become part of the official OpenCV-contrib repository[2]. This should secure the long-term sustainability of the CVVisual module. Furthermore, we hope that it initiates more research into tools for the iterative development phase of computer vision algorithms. At the same time, CVVisual should provide a solid basis and offer reusable components for direct extensions that cover additional OpenCV modules. The CVVisual documentation is also part of the current OpenCV online documentation[3]. A tutorial can be found within the CVVisual repository[4] and is also available online[5].

The remainder of the paper will first give an overview of research into visual debugging. Section III addresses the design considerations behind the CVVisual library. In section IV the core functionality, i.e. visual debugging and development tools of CVVisual, is detailed together with the background methodology. Section V concludes the paper with a brief summary and provides an outlook at future research.

II. Visual Debugging and Debugging of Vision Algorithms

The origins of software visualization reach back to the 1980s, when it became feasible to provide software developers with a graphical representation of their intangible subject matter [3]. One important motivation for this line of research is going beyond the linear structure of program code. The goal is to complement symbolic languages with graphical elements in order to improve human understanding. Visualization in general can be described as "the need or desire to see phenomena and relationships in new or different ways" [4]. In case of software there is often a wide gap between what is happening in the system and the perceivable effects. For example, if the result of a deep image processing pipeline is only the label of some (supposedly) recognized scene objects, many steps separate cause and effect. Although the intermediate steps are known, since they are written down as the program code, their contribution to the final result is largely dependent on the input data. Thus, their principal function is known, but their concrete effects often have to be empirically determined during development. For this task, visualizations provide alternative perspectives for each step, thereby helping to identify problems more easily. Furthermore, software visualization does not only help experienced developers to understand the structure and behaviour of software, but also assists students in learning about software development and programming [5].

Visual debugging of static and dynamic program structure as well as data and control flows is one important subfield of software visualization. Once more, the goal is to aid the developer in understanding the software, yet, here with a special focus on finding and correcting program errors. To a large extent, the means to this end are visualizations that provide information about the program in a non-textual or non-numeric format, such as graphs, diagrams, maps and images (cf. [6]). Most tools provide some form of interactivity, which enables the developer to focus on the aspects relevant to him and filter out others. The second element is often to provide different perspectives on the same information. A basic example is a tool that provides a tabular view of the data and a graphical plot of the same data items. A very literal example of multiple perspectives can be found with regard to the visualization of oceanographic data [7]: Each scientist can inspect the data from different perspectives in a CAVE virtual reality environment. However, even in this example "perspective" does not only refer to the spatial point of view, but also to the way each data item is graphically represented.

Compared with the various tools available for visual debugging, there is a shortage of tools when it comes to dealing with debugging images – perhaps the most visual data type. To our knowledge no generic tool exists that explicitly addresses

[2]https://github.com/Itseez/opencv_contrib
[3]http://docs.opencv.org/trunk/
[4]https://github.com/Itseez/opencv_contrib/tree/master/modules/cvv/tutorials
[5]http://cvv.mostlynerdless.de/

the challenges faced by the developers of computer vision software. The tool with the closest resemblance to CVVisual is the Image Watch[6] for Visual Studio. If a OpenCV program is run in debug mode, Image Watch enables to view the images that are currently in memory without adding code to the application. However, the only available option is to separately view each image. Therefore, all visualization features provided by CVVisual (e.g. Fig. 6,7) would require the same custom debug code, which is required without Image Watch. Probably, many, if not all of the visualizations currently contained in CVVisual, have existed before. Nonetheless they have probably not been generic, interactive and part of an integrated toolset, but were more similar to code in Fig. 1. CVVisual as interactive visual debugging tool for computer vision applications will be described in the following section.

III. Library Design

The CVVisual module was designed and implemented with three main objectives:

- First, CVVisual must be as generic and portable as the OpenCV library which it targets.
- Second, the debug code must be unobtrusive and very brief.
- Third, if the source code containing CVVisual calls is compiled in release mode, the program must not have additional dependencies and there must be no performance or memory penalty introduced by the CVVisual calls.

The first objective was achieved by depending only on the C++ standard library, OpenCV itself and one of its optional dependencies, namely Qt[7]. As will be explained for the third objective, Qt is a dependency only in debug mode but not in release mode. Accomplishment of the second objective is demonstrated by Fig. 4: Accessing each visualization type only requires a single function call with the required data and further optional items as arguments. Objective three is addressed by a clear separation of interface and implementation in combination with use of preprocessor macros in the application visible header files. Besides of fully removing the CVVisual calls for each compilation unit, one can also temporarily deactivate them by a function for some parts of the code. In this case CVVisual is still part of the program, but the disabled calls will only have a negligible performance impact.

Apart from these main objectives, we aimed for a clean and modular implementation. This concerns the separation between interface and implementation, the loose coupling between API and GUI and the compositional approach to each GUI view. Instead of directly calling into the GUI code, the implementation and its dependencies are hidden from the application code through a dependency free function call. In order to provide metadata about a CVVisual call, e.g. on which line of which file and in which function it appeared, all calls take an optional `CallMetaData` object that is passed to the GUI. The `CallMetaData` can be conveniently generated with a macro provided by CVVisual (see Fig. 4).

[6]http://go.microsoft.com/fwlink/?LinkId=285460

[7]http://www.qt.io/

```
cv::VideoCapture capture(0);
cv::Ptr<ORB> detector = cv::ORB::create(500);
cv::BFMatcher matcher(cv::NORM_HAMMING);
for (int i = 0; i < 10; i++) {
 capture >> imgR;

 std::string iname{"imgR" + toStr(i)};
 cvv::showImage(imgR, CVVISUAL_LOCATION, iname);

 cv::cvtColor(imgR, imgG, CV_BGR2GRAY);
 cvv::debugFilter(imgR, imgG,
                  CVVISUAL_LOCATION, "gray");

 detector->detectAndCompute(imgG, noArray(),
                            kpnts, descrs);

 if (!pImgG.empty()) {
  std::vector<cv::DMatch> matches;
  matcher.match(pDescrs, descrs, matches);
  cvv::debugDMatch(pImgG, pKpnts, imgG, kpnts,
                   matches, CVVISUAL_LOCATION,
                   mname);

  std::sort(matches.begin(), matches.end());
  matches.resize(numFilteredMatches);
  cvv::debugDMatch(pImgG, pKpnts, imgG, kpnts,
                   matches, CVVISUAL_LOCATION,
                   bmname);
 }
}
cvv::finalShow();
```

Fig. 4. Excerpt of example code using different CVVisual calls. The program captures images (a), converts them to grayscale ones (b) and detects keypoints and extracts descriptors for them. If a previous image is available, the descriptors are matched to those of the previous image (c). Of these matches, the best ones are kept (d). For each step (a)-(d) the intermediate results are passed to CVVisual for interactive inspection.

The metadata can be used in the overview window to sort and filter the debug information (Fig. 3). The GUI widgets are divided into utility and application widgets. The former provide generic functionality, such as a zoomable and syncable image display. The latter widgets are directly related to a specific visualization perspective (see Sect. IV). A further modularization is implemented in the split between option panels and their related display widgets. This allows for a more flexible reuse of these widgets because control elements do not interfere with the visualization. For example, the widget to display a single image (Fig. 5) is instantiated multiple times in other views (Fig. 7) with each control panels in a separate area.

If a developer requires a custom visualization perspective or debug call, he can either directly add it to the source code of CVVisual or register it as a plugin. The first option has the advantage that the new perspective becomes part of the CVVisual module and can be easily contributed to the community. Yet, it requires a recompilation of the whole module. If the second option is chosen instead, no recompilation is required and the new perspective or debug call is still available in the same manner as the preexisting ones. Due to the availability of reusable widgets, writing a plugin does not require much more code than creating a static output of the data (see Fig. 9). Having described the technical background of the CVVisual module, the next section looks at it from the perspective of the user, i.e. the computer vision developer.

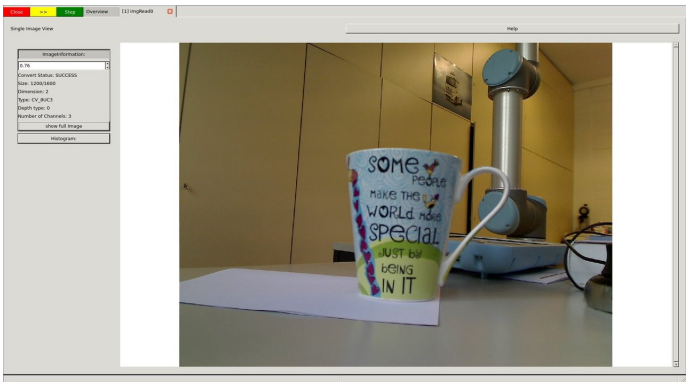

Fig. 5. This figure shows the single image view, which is related to data added by `cvv::showImage`. It has very similar features to `cv::imshow` provided by OpenCV highgui module. However, in contrast to the many unrelated highgui windows, all calls are listed in the overview widget and can be closed and opened in any combination from it (cf. Fig. 3).

IV. Interactive GUI

The library and API provide the basis which facilitates our approach to the iterative development phases of vision applications. Nevertheless, the actual benefits, the flexible interactive visual debugging, are provided by the CVVisual GUI. We will describe its main functionalities and currently available visualizations following the user visible events of the code in Fig. 4 compiled in debug mode.

The first time a `cvv` call is encountered during execution of the program, the CVVisual library and GUI is automatically initialized. By default each `cvv` call blocks the main program and control is transfered to the CVVisual GUI. The GUI will display the newly added data in the overview window (Fig. 3). All previous data remains available throughout the program runtime and can be accessed through the overview window at any time. The data can be sorted, filtered and grouped with the flexible filter engine available from the input field at the top. In our example, the first call is `cvv::showImage`, which adds a single image to CVVisual. The first optional arguments of all `cvv` calls is the `CVVISUAL_LOCATION` macro, which creates a `CallMetaData` object (cf. Sect. III). The second optional argument is a string to describe the call. It can be used in the overview to find images or sort them better in a large program with many CVVisual calls.

The view related to `cvv::showImage` (Fig. 5) is similar to `cv::imshow` and features zoom down to the level of pixels. At the highest zoom levels the image is overlaid with the numerical pixel values. CVVisual provides three options to continue the program: Step by Step, continue until `cvv::finalShow` or continue the program without further CVVisual interaction. The first option runs the program until the next `cvv` call. The second option adds all calls to CVVisual, but does not block and show the GUI until `cvv::finalShow` is encountered. Due to the fact how Qt applications are handled in the background, a `cvv::finalShow` must appear before regular termination of the program. This can also be ensured by a included Resource Acquisition Is Initialization (RAII) style class.

If in our example continue “Step” is chosen, the GUI will be hidden until `cvv::debugFilter` is encountered. This

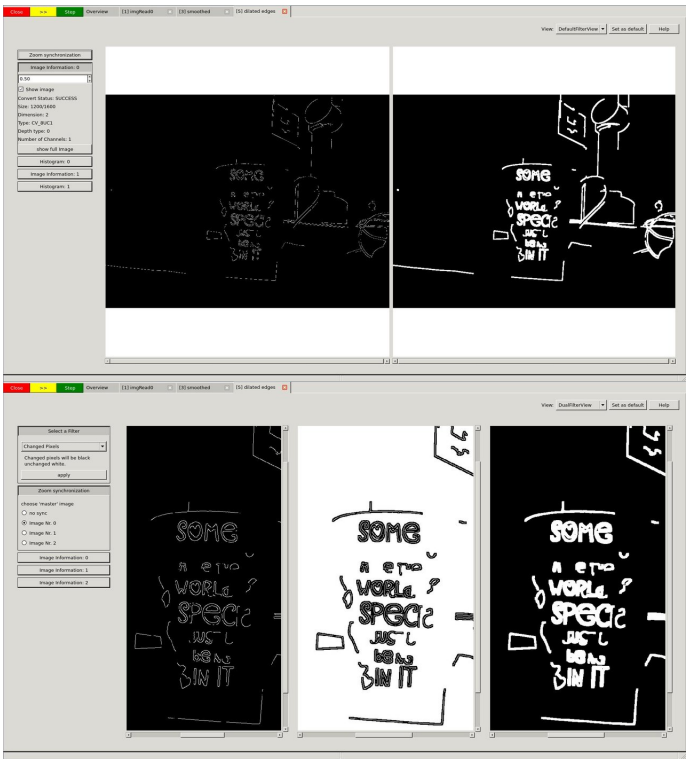

Fig. 6. The two screenshots show different perspectives of the filter view for `cvv::debugFilter` calls. The upper perspective provides an interactive and optionally synchronized side by side view of the image filter input and output. In the lower perspective, the differences between the images are shown. Multiple options to display the pixel differences are available, such as changed pixels and difference images. Further perspectives include an overlayed visualization of input and output image with various interactive display settings.

call takes two images as parameters. Its semantics are related to image filters in the broad sense: The second image is supposed to somehow originate from the former one – e.g. through a filter operation. The difference becomes more clear through the GUI. If an item added by `cvv::debugFilter` is selected in the overview the two images are shown in the filter view (Fig. 6). The filter view provides different perspectives or visualizations of the two related images. The first and simplest perspective is to show both images next to each other. Each image can be either zoomed independently or the visible image regions can be synchronized. A similar basic perspective is alpha blending of the two images. More complex perspectives show the pixelwise difference between the images in various forms. In order to more easily identify the *relevant* information about the result of applying the filter, further post-processing options are available (e.g. Fig. 9).

Continuing with the example code in Fig. 4, the next call is `cvv::debugDMatch`. Its parameters are two images, their keypoints and a calculated match between these. On selection of this call from the overview, the match view is shown (Fig. 7). Due to the complexity of dealing with the details – and possible errors – of descriptor matching, this is the most elaborate view in terms of features. Two different perspectives are currently implemented: The line match perspective and the translation match perspective. In the line match perspective each matched pair of descriptors is connected by a line. Since

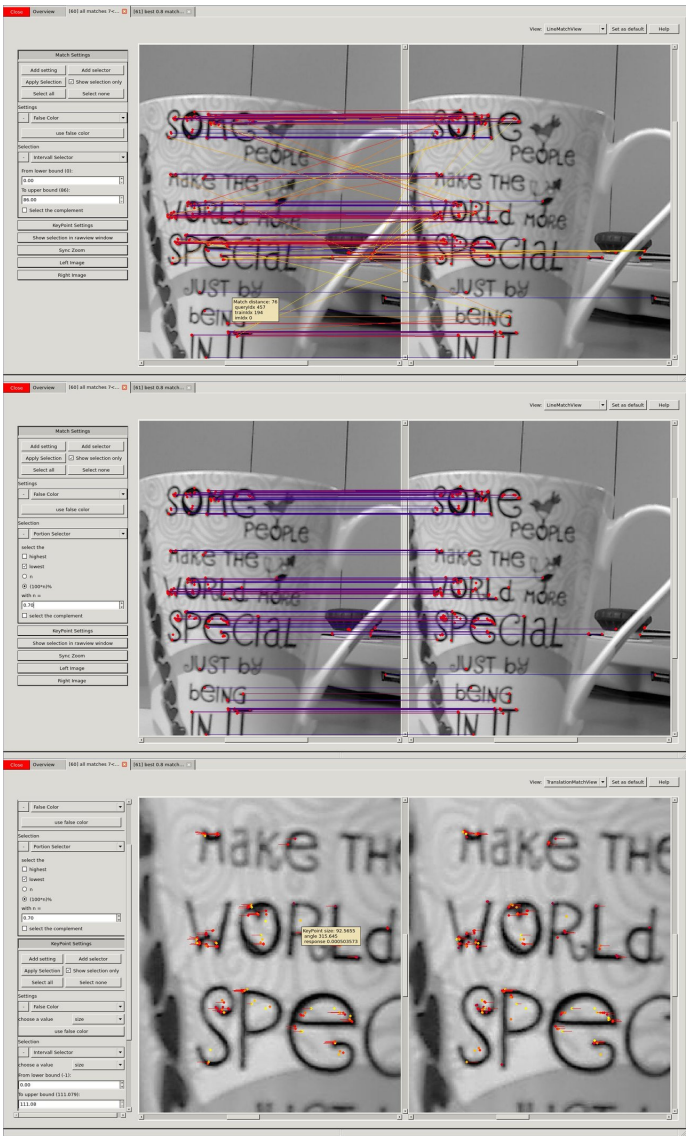

Fig. 7. Two perspectives of the match view for `cvv::debugDMatch` calls are shown. The upper two screenshots visualize matches in a similar way to the OpenCV `cv::drawMatches` function. However, they are not static images. Rather they are interactive visualizations that also provide mouseover information about each individual match. In addition – as one can see by the difference between the top and middle screenshot – it is possible to filter the matches interactively. For example, some best or worst percentage of matches in terms of their match distance can be selected. The bottom screenshot depicts an alternative visualization of the relation between matched features.

the lines are interactive elements, all interaction available in the single image view (Fig. 5) is also available here. It is possible to zoom into the images, either independently or synchronized. The lines are automatically updated and adjusted to the visible image region. Depending on the image content, different line colors are favorable for good visibility. The color can be either a freely chosen single color or they can be colored according to the quality of each match. The translation perspective visualizes the relation between the matches in a different manner. For each match a line originating in the descriptor of this image is drawn within the same image to the position of its match in the other image. The user can chose colors as before. Many filter options are available to select subsets of the matches to be visualized. Both perspectives also offer context information and visualization options for the descriptors.

In addition to the views, which are specific to a `cvv` call, there is also the generic raw view (Fig. 8). The raw view is a tabular perspective of the images and matches seen as numerical data. It features the same filter engine as the overview window. Furthermore, it can be used in combination with other views to manually select only specific data items for visualization. An additional capability of the raw view is to export all or a selection of the data in different output formats.

As exemplified in Fig. 9, all views can be easily extended by additional elements. A GUI extension can be either implemented directly or as an external CVVisual plugin (cf. Sect. III).

V. Conclusion

We presented an OpenCV module for the interactive visual debugging of computer vision applications. Our goal was to aid the programmer during iterative stages of development by providing visual tools. The use of these tools is intended as a replacement for the current practice of writing custom ad-hoc code in a printf-debugging manner. Moreover, our CVVisual module is a generic, interactive and integrated tool, which offers alternative perspectives of the underlying image data.

CVVisual already has the functionality to cover a large part of daily computer vision debugging needs. Yet, perhaps more importantly, CVVisual provides a solid basis that facilitates the integration of future research in software visualization for computer vision developers. From a practical point of view, an important next step is to extend the coverage of visual debugging to other similar OpenCV modules such as stitching and superres. Although it is important and would provide large benefits to come up with suitable perspectives for these and other image processing modules, another direction of research might have an even bigger impact. There is also a line of research on visualizations in the machine learning and data mining communities (cf. [8]). The OpenCV machine learning module (ml) is likely one of the most often used frameworks for applied machine learning, especially if the application is also related to computer vision. If CVVisual would be extended by interactive visualizations that combine the results presented here with those in the machine learning community, this would greatly improve the development process of OpenCV-related applications such as robotics.

Acknowledgment

This research was carried out with the support of the German Research Foundation (DFG) within project I05, SFB/TRR 125 “Cognition-Guided Surgery”. The authors also want to thank Johannes Bechberger, Erich Bretnütz, Nikolai Ganer, Raphael Grimm, Clara Scherer and Florian Weber for their essential contribution to CVVisual.

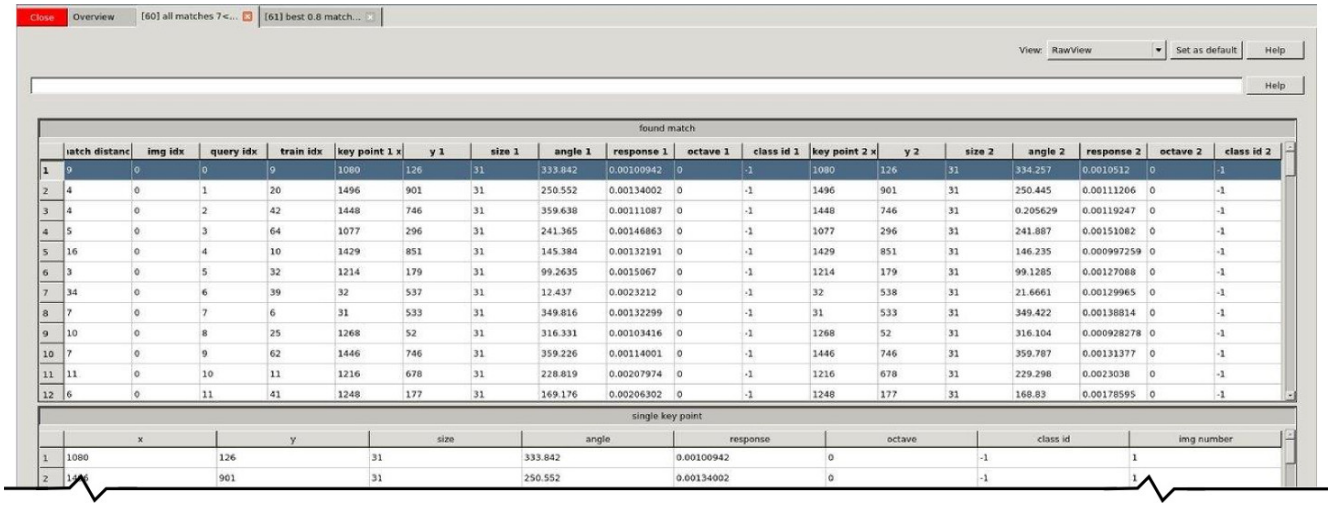

Fig. 8. The CVVisual raw view was selected in this screenshot. It is available in adapted variants for all types of `cvv` calls. Besides providing all the exact numerical information about the images in a sortable tabular format, it also features the same powerful search and filter engine as the overview window (cf. Fig. 3).

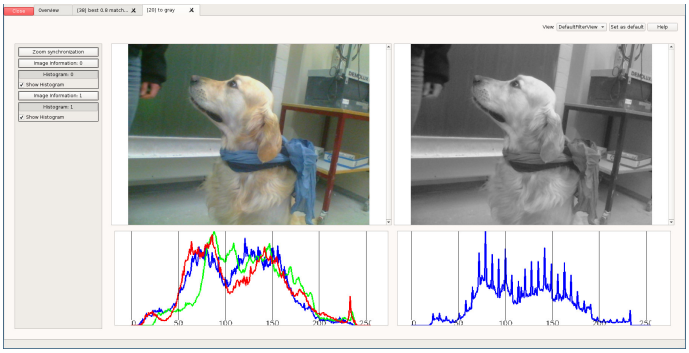

Fig. 9. CVVisual was designed and implemented with the goals of modularity, reusability and extensibility in mind. The CVVisual GUI widgets can be either reused as components or extended through inheritance. Furthermore, a plugin system is in place to allow the extension of the GUI by developer-provided views and perspectives without having to recompile the core module. One example of an extension is the interactive image histogram for the filter view shown in the screenshot. Its implementation only consists of 200 lines of code of which about half is already required to calculate and draw the color histogram.

References

[1] A. Bihlmaier and H. Wörn, "Automated Endoscopic Camera Guidance: A Knowledge-Based System towards Robot Assisted Surgery," in *Proceedings for the Joint Conference of ISR 2014 (45th International Symposium on Robotics) and ROBOTIK 2014 (8th German Conference on Robotics)*, 2014, pp. 617–622.

[2] ——, "Learning surgical know-how: Dexterity for a cognitive endoscope robot," in *Proceedings of the 7th IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and the 7th IEEE International Conference on Robotics, Automation and Mechatronics (RAM)*, 2015, forthcoming.

[3] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211 – 266, 1993.

[4] M. Friendly, "A brief history of data visualization," in *Handbook of Data Visualization*, ser. Springer Handbooks Comp.Statistics. Springer Berlin Heidelberg, 2008, pp. 15–56.

[5] Y. Ding, Y. Hang, G. Wan, and S. He, "Application of software visualization in programming teaching," in *Computer Science Education (ICCSE), 2014 9th International Conference on*, Aug 2014, pp. 803–806.

[6] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in eclipse," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange*, ser. Eclipse '07. New York, NY, USA: ACM, 2007, pp. 31–35.

[7] K. S. Park, A. Kapoor, and J. Leigh, "Lessons learned from employing multiple perspectives in a collaborative virtual environment for visualizing scientific data," in *Proceedings of the Third International Conference on Collaborative Virtual Environments*, ser. CVE '00. New York, NY, USA: ACM, 2000, pp. 73–82.

[8] J. Talbot, B. Lee, A. Kapoor, and D. S. Tan, "Ensemblematrix: Interactive visualization to support machine learning with multiple classifiers," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 1283–1292.