Building Modular Vision Systems with a Graphical Plugin Environment

Frank Lömkör, Sebastian Wrede, Marc Hanheide, and Jannik Fritsch
Applied Computer Science, Technical Faculty, Bielefeld University,
P. O. Box 10 01 31, 33501 Bielefeld
{floemker,swrede,mhanheid,jannik}@techfak.uni-bielefeld.de

Abstract

With the increasing interest in computer vision for interactive systems, the challenges of the development process involving many researchers are becoming more prominent. Issues like reuse of algorithms, modularity, and distributed processing are getting more important in the endeavor of building complex vision systems. We present a framework that allows independent development of enclosed components and supports interactive optimization of algorithmic parameters in an online fashion. The communication between components is performed nearly without any slow down compared to a monolithic system. Through the modular concept, all components can be flexibly distributed and reused in other application domains. The suitability of the approach is demonstrated with an example system.

1 Introduction

In recent years there is an increasing interest in realizing complex vision systems suitable for human-machine interaction. Different from research on individual algorithms that are usually developed and tested offline, systems for use in an interaction scenario need to be capable of real-time operation. While this requirement could be seen as primarily related to the algorithmic implementation, today’s complex vision systems consist of a number of algorithms that achieve the task at hand by adequately combining the individual processing results. Consequently, the distribution of the different algorithms across different computing nodes is an important strategy to achieve real-time performance.

Although such a distributed operation of a vision system is beneficial for an interactive system, the real challenge in research on complex vision systems is the development process involving many researchers. In this contribution we present an integration framework for the development of vision systems that addresses aspects that we found crucial for successfully building large vision systems. We consider the following requirements to be highly important: ease of use, online parameterization of algorithms, flexible visualization of processing results, simple integration of 3rd party libraries, rapid prototyping through reuse of existing algorithms.

Although this list does contain typical requirements, there seems to be no vision-related graphical development environment adopted by the vision community at large. There have been many attempts (e.g., [15], [1]), but often a computer vision researcher starts to build his own environment when thinking about a new algorithm or system, e.g. when starting his PhD thesis. Not only time is invested in this moment, but also problems arise when the system starts growing and needs to run with higher performance or needs to be integrated in a larger system architecture. Then it is very likely that the ad-hoc solutions which perfectly fitted personal requirements do not meet the additional requirements stemming from the task of integrating a complex vision system.

Thus, we present a graphical plugin shell and integration framework that saves development time even for simple applications and can grow with the increasing complexity of a sophisticated computer vision system. The reuse of existing algorithmic solutions provided by other researchers is supported by the plugin concept while the requirements of complex vision systems are met by the possibility of ‘splitting’ a processing chain into several processes running on different machines. In the following we will give a detailed description of the iceWing framework that is provided as open source [12] and how it supports vision researchers in developing complex modular vision systems.

2 Related Work

Both, in the scientific as well as in the commercial domain a broad variety of available software exists related to vision processing. This starts with popular computer vision libraries, e.g. OpenCV [9] or RAVL [4], that contain many algorithms and data structures useful in building vision system. Although those libraries are widely accepted and speed up the development process of vision algorithms,
Figure 1. A typical session with iceWing. Various intermediate results are being displayed and could be examined. Various parameters can be influenced interactively. A description of the single parts can be found in the text.

they provide no or minimal control and integration patterns for the construction of vision systems [25] and only limited visualization and parameterization capabilities.

The ImaLab system [14] is an example for a more sophisticated vision development environment which allows different modules to be developed and integrated in an interactive manner. It utilizes a C++ interpreter to support rapid prototyping of new algorithms and gives immediate feedback on parameter changes to the developer.

Following the idea of prototyping vision algorithms there are many different fully graphical interactive image processing environments available where an often predefined set of operators can be combined to construct new algorithms. This often commercial class of software ranges from rather small packages like Ad Oculos [18] over products like HALCON [15] to enterprise class software like VisiQuest [1] (originating from the popular Khoros image processing environment [11]). Besides that those tools are mostly commercial, the integration of already existing code is often a tedious task.

A widely used general prototyping tool is MATLAB [19], which provides good visualization and user interface generation possibilities as well as a large number of internal and external library functions for different tasks. But using it directly in complex vision systems is often problematic because of speed deficits. Moreover, its proprietary license can easily interfere with distributed development, as every participating partner needs an own license.

Except the high-end commercial tools mentioned above, the freely available toolkits mainly have one of the following two drawbacks. Either they provide only very generic support for the development of vision algorithms, e.g. low-level image processing libraries like OpenCV, or they are very specialized in one sense, e.g. for rapid prototyping or educational purposes. Even though the ability to prototype should be considered as very important, a framework or tool should not complicate the step from single algorithms to vision systems.

From the perspective of building integrated vision systems there are even more questions that need to be addressed. To give an example, one important question is the openness of those legacy toolkits. This describes the ability...
to interface and to integrate the developed algorithms within a larger vision system architecture.

In the remainder of this contribution we will describe our plugin-based vision processing framework that allows for fast development of algorithms and scales well with increasing requirements towards the integration of developed image processing modules in a larger vision system.

3 The iceWing architecture

iceWing is a graphical plugin shell optimized for the special needs arising in the field of vision system development. In this area enclosed development of single components as well as flexible communication and interaction between these components is highly relevant. At the same time the encapsulation and communication overhead should only have a negligible impact on the overall execution speed. This is important if small data elements are exchanged between single components but in particular for large data elements such as images.

At the same time the framework should help to develop and optimize the single components as well as the integrated system. For this easy and fast introspection of any intermediate results as well as easy and flexible modification of algorithm parameters at any time are key points. As will be seen in the next subsections, all these requirements are well covered by iceWing.

3.1 Plugin interaction

Figure 1 shows a typical session with iceWing. The overall architecture of iceWing is shown in figure 2. iceWing itself only provides an administrative core, an initially minimal user interface and a variety of support functions for tasks like user interface creation and communication. The real functionality for the task the user wants to solve with iceWing is provided by dynamically loaded plugins, which are realized as standard shared libraries. For the user plugin development is as easy as deriving from an abstract base class and implementing a process method that carries out computations on data passed to the method by iceWing. During these computations plugins can take advantage of other external libraries, for example OpenCV or RAVL, for enhanced image processing functionality. Plugin development can be done in C and C++. Additionally, bindings for the scripting languages Python and MATLAB are currently developed.

For communication between these plugins two distinct communication patterns are provided, a data driven data storage/data observer interface similar to the known Observer pattern [6] and a function storage and retrieval interface for procedural communication. Data items for the first communication form are represented in iceWing by a reference to the real data and an identifying string. This allows to exchange any data without any restrictions and without the need of any preprocessing. Plugins can observe the storage of any number of such data items. If an observed data item is stored by a different plugin, the observing plugin gets called with the new data after the storing plugin has finished its work. The data itself is not copied during the complete process, but managed by reference counting, allowing a fast communication between plugins even for large images. The now running plugin may again store any number of new data items with equal or different identifiers and thus initiate again the call of other plugins. Depending on the identifier, this might be plugins which were not called beforehand or ones already called before for another data item. Initiated is this process by iceWing by storing a data element with the special identifier “start”. The architecture figure 2 shows this processing loop on the left side.

Normally, data is passed to observing plugins based on the identifier of the data regardless of which plugin stored the data. But additionally, the user can select in the user interface at runtime from which plugins an observing plugin should be able to receive data. Thus a dynamical filtering on a functional level is provided.

Besides this data driven communication pattern the plugins can provide any number of C functions under different identifiers. Afterwards other plugins can retrieve and call these functions. These calls, again, do not impose any noticeable performance slow down compared to a traditional monolithic program. All these different communication patterns are normally serialized, i.e. they happen one after another. Of course, the plugins are free to start new threads and thus perform any calculations in parallel. At the same time more advanced and more dynamical distributed communication patterns are possible. This will be described in detail in section 4. For real time introspection of the plugin processing different statistic views and a logging view are provided in the user interface (Fig. 1a).

3.2 User interface

Especially during development of algorithms but also during use of the final system visualization of different intermediate results is often needed. Hence iceWing enables the plugins to create any number of windows where the plugins can display any results (Fig. 1b). The display of images with various bit depths, data types, and color models and the rendering of text and various vector graphic objects is supported. Optionally, plugins can extend the render pipeline with functions for rendering new object types. Only if the user opens a window, time or memory for the rendering operations is consumed. So the plugins can create several windows to visualize different internal states
without speed deficits. All interactions with the windows, for example zooming, panning, changing rendering parameters like fonts, introspecting the image, or saving the window content optionally continuously to disk (Fig. 1c), are completely handled by iceWing. Redrawing the complete window content, including any vector drawings, is done completely autonomously. As shown in figure 2 this is done asynchronously to the main processing loop in an own thread. As nearly everything is handled by iceWing, on the one hand displaying results gets easy for plugins and on the other hand the user has for all windows always full control over the rendering and introspection of the results.

Of at least the same importance as the visualization is the flexible parameterization of algorithms. iceWing provides for this different easy to handle user interface elements, for example sliders, lists, or entries in context menus in image windows (Fig. 1d). During the creation of the elements the user provides besides some meta information like minimal and maximal values for sliders the address of a variable. The basic idea behind the user interface is now to handle the complete interface completely by iceWing in the background in an own thread, while the main algorithm, realized via the interaction of different plugins, is working at the same time concurrently in the foreground. The plugins do not need to check any mouse actions on sliders or similar events. If needed, iceWing simply sets the variable, which is given during creation of the interface element, to the new value asynchronously in the background. Additionally to the simplified interface element handling, this provides instant apply of all values and enables completely autonomous and automatic support for loading and saving of all widget settings to/from different configuration files at any time during program execution.

In iceWing the idea of automatic configuration handling is taken one step further. Additionally to autonomous loading and saving of the widgets, the different widgets can be set at any time remotely from a different computer in a network. This enables for example simple synchronization of different programs in a network without any special programming effort from the plugin author as well as script based control of any running iceWing instances.

4 Using iceWing for the construction of vision systems

As motivated above, iceWing helps not only in developing and testing single algorithms but also provides an environment for building more complex vision systems. These systems may consist of single iceWing processes or multiple iceWing instances interconnected for increased processing performance that can easily be integrated into larger systems architectures. The necessary steps to achieve the level of transparent interoperability needed with regard to inter-module data exchange as well as the ability to increase processing power and integration into larger vision systems will be discussed in the following.

4.1 Building cooperative plugins

Although iceWing can handle any type of data between plugins, the internal data exchange interface does not allow us to guarantee the interoperability of arbitrary plugins. As one key to interoperability is low coupling, we thus decided on a two-fold strategy for data to be exchanged within the framework.
Images and other binary large objects are directly addressed by the different plugins and are thus not copied within one processing cycle. To provide a consistent data model, a library for the most common types of binary data and wrapper functions for the OpenCV image data types are provided in iceWing.

In contrast to this, a wide variety of symbolic, vision related data structures, e.g. region or object descriptions are transmitted as structured, labeled data between plugins following the XML data model. It is accessed using an API based on the XPath [2] language which can navigate on this data both by node names and by their content. The rationale is that by using a path language varying structures can be handled without changes to existing code, avoiding versioning issues effectively yielding low coupling. Having a set of XML data types for image analysis and vision processing, plugins can be written that work only on common subparts of transmitted XML data.

For instance, figure 3 shows four plugins processing partially equivalent XML data structures. This allows a plugin (e.g. "Visualize Tracks") to observe data from different other plugins. Only the part in the XML document that contains information about center points ("<CENTER>") has to be present in an exchanged data item. Starting with a simple partial path specialization to access the context node, e.g. in this example as simple as "//*/CENTER"", the extraction of contained information is easily possible although the context node itself might appear in varying places of XML structures. Thus, this path expression e.g. works on both data structures shown in figure 3. As long as no necessary information is removed, this strategy yields loose coupling and interoperability between separately developed plugins.

Hence such plugins, named cooperative, can be used in every iceWing system that follows these design principles. That said, every module developer commits himself for the development of his plugins to be cooperative in terms of plugin reuse. As soon as the first plugin can be reused by other vision system developers, the effort for familiarization with the iceWing framework and the XML/XPath data encoding scheme begins to pay off by itself. With a library of plugins even complex vision systems as described in section 5 can almost be constructed of the shelf and customized to specific application needs without having to recompile a line of code.

4.2 Transparent distribution and integration with XCF plugins

Having a large library of cooperative plugins not only pays off in higher reuse of plugin implementations but also narrows down the gap between vision researchers on the one hand and system integrators on the other hand.

To achieve this, we provide two infrastructure plugins termed XCFImport and XCFExport that extend the vision processing framework transparently by features for parallel and distributed operation. With those plugins data can be collected from or published to several different iceWing instances running on an arbitrary number of networked standard computers. Those plugins need in general only be configured with the identifier corresponding to the data that should be exported or the identifier under which the imported data will be made available to the registered plugins. Neither a special meta-compiler or data-description is necessary to make use of those plugins nor any implementation change of existing plugins is needed.

The ability to distribute algorithms over several computing nodes is especially useful when after the initial tests and algorithmic optimizations performance is still an issue. Boosting performance is often possible through distributed computing by simply using the infrastructure plugins together with the set of cooperative plugins developed by the vision researcher. In section 5 a detailed example will be given how an application benefits from this approach in terms of performance gain.

Furthermore, the infrastructure plugins allow for an easy integration of iceWing processes as a sub-system into larger system architectures. The infrastructure plugins utilize the open source XCF SDK [23] for data- and event-driven system integration [24], which has been developed recently in the VAMPIRE [20] cognitive vision project. Note, that those plugins can be easily exchanged by other infrastructure plugins if needed. In our view, however, XCF is well suited for this task because it itself makes extensive use of XML and equally allows for native transmission of binary data in an approach similar to the one proposed by the W3C XOP recommendation [22]. Within VAMPIRE and the XCF SDK, the idea of partial path specialization to access parts of XML documents has already been used successfully to build loosely coupled system architectures.
All those integrative features come at almost no cost for the module developer and are part of the iceWing concept. Following those guidelines outlined above, reuse is facilitated and system integration with the described plugin concept can be carried out in a very efficient manner. How this is done for a concrete example is described in the next chapter.

5 Exemplary plugins and applications

iceWing is already utilized in many different vision applications. These range from simple camera configuration or camera grabbing to much more complex integrated systems like recognition of manipulative gestures [5] and interactive object learning [8, 13]. To give an impression some realized plugins will be listed below and one of the more sophisticated systems will be presented in greater detail as an exemplary use case for the integration of vision systems with iceWing.

5.1 Plugin overview

A wide variety of plugins have been developed in the course of the different applications, ranging from general ones to project specific ones. One of the fundamental plugins allows reading of image streams from sources like multiple disk files, movie files, and various grabbers and cameras (that is: firewire, V4L, V4L2, IIDC 1394, and, on Digital UNIX systems, MME). With the infrastructure plugins introduced in section 4 iceWing instances can communicate in a network. A firewire configurator can modify camera parameters on the fly. Different segmenter such as mean shift [3] and color structure code [16] are realized as individual plugins. A queue of different basic filter operations, e.g. image smoothing, image cropping, or color conversion, can be applied to images with one plugin. Two plugins allow to represent homogenous regions from images as polygons and to track those polygons. Some alternative tracking plugins work directly on images and track objects based on directly calculated image features. A Viola & Jones [21] based detector plugin allows to recognize faces and objects. Action classification is done with a Condensation based classifier [10] applied to hand trajectories. Three plugins for a mosaicing task will be described in more detail in the next subsection.

5.2 Exemplary application: Mosaics of planar sub-scenes

The application chosen to explain integration aspects and evaluate distributed computing with iceWing creates mosaics from an arbitrarily moving stereo camera pair. This use case is expected to benefit a lot from distributed processing and comprises individual algorithms developed by different researchers and integrated as cooperative iceWing plugins. The following presentation focuses on the integration aspects of the system. Details of the underlying algorithms can be found in [7].

Figure 4 depicts the architectural sketch of the three stage system along with some results of each stage. The two cameras mounted to a helmet that realizes an augmented reality device [17] (see upper left of figure 4) capture stereo images. The iceWing-Plugin Feature Point Tracker then detects feature points in the left image and tracks these at about 15Hz frame rate. Another plugin Plane Detector (cf. figure 5) computes stereo matches from the tracked feature points and the corresponding right images. In terms of iceWing the plane detection observes stereo images. Two plugins allow to represent homogenous regions from images as polygons and to track these polygons. Some alternative tracking plugins work directly on images and track objects based on directly calculated image features. A Viola & Jones [21] based detector plugin allows to recognize faces and objects. Action classification is done with a Condensation based classifier [10] applied to hand trajectories. Three plugins for a mosaicing task will be described in more detail in the next subsection.
can not be done at frame rate, parallel processing of tracking and plane detection is essential. Once planes are detected by stereo matching, tracking feature points indeed allows to track the individual planes. Finally, individual mosaics are asynchronously computed whenever new plane information becomes available.

5.3 Distributed system

In the presented use case, computational effort is relatively high for each plugin of the system. In particular it is essential that feature point tracking is running at high rates for not loosing tracked features. As stated before, it is transparent for the developers of plugins whether all processing is performed in one instance of iceWing or distributed over several computational nodes (see Figure 4). Here the main processing plugins (indicated by shaded boxes) are enframed by communication plugins which provide transparent network transport of XML data as well as referenced binary data. To evaluate the benefit of distributed programming in this use case, we compared the performance of the distributed system running on three machines (2x Pentium 4 2.4 GHz, SPECint2000≈880; 1x Pentium 4 2.0 GHz, SPECint2000≈750) with a system running on just one computer (Pentium 4 2.0 GHz). Figure 6 shows performance results in terms of seconds per execution cycle. Here the benefit of distributing the different plugins across three computational nodes becomes obvious. While on a single computer it takes more than 15 seconds until a current image is finally integrated into the mosaic, this time drops to less than 5 seconds when distributed. This time speedup corresponds well to the increased computing power.

6 Conclusion

Building complex vision systems emphasizes already known challenges and especially brings new challenges to the vision community. We presented a modular plugin environment, which covers these challenges and thus greatly enhances the productivity of developers.

Consequently realizing the different processing stages of the vision system as iceWing plugins allows to test and evaluate them individually and facilitates easy integration of the already tested components. By using binary data for large data objects and XML data with a flexible XPath-querying scheme for the remaining ones, fast as well as very flexible communication is provided. The achieved flexibility is essential for building real vision systems. By using the graphical facilities, fast, easy, and very direct optimization and introspection of the involved algorithms gets possible.

The software packages, iceWing as well as XCF, are available as open source licensed under the GPL. Those interested in using them can download the software at [12] and [23].

References

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.


