libugrab - A Versatile Grabbing Library

Ferenc Kahlesz Reinhard Klein
Computer Graphics Group
University of Bonn
53117 Bonn, Germany
{fecu,rk}@cs.uni-bonn.de

ABSTRACT

Current commercial and freely available grabbing libraries are tightly coupled to operating systems and/or imaging hardware. Moreover, they usually do not support any kind of distributed camera-systems. This forces developers to either reimplement significant parts of the application or to come up with elaborate abstraction for the grabbing, should the underlying operating system, hardware (e.g. changing from analog PAL sources to IIDC cameras) or distribution model (e.g. adding remote intelligent cameras, which are capable of image processing themselves) change. In this paper we describe 'libugrab', a versatile grabbing library designed to provide a flexible abstraction of the grabbing process. The main advantages of 'libugrab' over similar libraries are the following: open source license, cross-platform availability, network transparency, support for both push and pull grabbing models, built-in support for image-processing via callbacks. The design especially facilitates rapid prototyping of distributed vision systems, which we demonstrate by several examples.

Keywords: image processing, middleware, grabbing, intelligent cameras

1 MOTIVATION

Applications that do image processing vary considerably with respect to the complexity of the software logic they need to acquire data from their imaging sources. Depending on whether a system has to do offline or real-time (RT) processing, whether it is monocular or multiview or whether it uses a single computer or computation is distributed among several nodes, the associated implementation efforts vary heavily.

1.1 Offline monocular systems

If online processing is not required and the system is monocular, the software infrastructure can be held simple: a standalone application records a video-file, which is later read by an actual processing program. Conveniently, the recording application might be built-in in the used operating-system or comes with the purchased camera. A skeleton-code for an application that loads the images from the recorded file and makes them available for processing can be implemented once and easily reused for different algorithms. This allows researchers to concentrate on the vital task at hand: developing the actual image processing algorithm. Examples for systems that can be developed this way are computationally expensive image segmentation or object recognition methods.

1.2 Offline multiview systems

Even if only the monocular requirement is changed to multiview, the software infrastructure can get considerably more complex. Although the complexity of boilerplate-code for multi-source offline processing is basically the same as for the monocular counterpart (and all the benefits described above still hold), this is not necessarily true for the image acquisition phase. This is due to the fact that the grabbed images usually have to be synchronized. Depending on the imaging source used, this can be done in hardware (e.g. genlock) or in software (e.g. adding timestamps to frames and selecting close ones later). Multiview algorithms benefit from using a higher number of cameras and researchers normally would like to experiment with different camera placements and numbers. This can lead to two acquisition difficulties: first, different off-the-shelf cameras might have different framerates and second, the desired number of cameras might prohibit the use of a single computer, due to harddisk and/or PCI-bus bandwidth limitations. In the latter case, the grabbing application should be written network-aware and relatively high-precision synchronization of the clocks of the involved computers is also necessary.

1.3 Online multiview systems

Among other research areas, the currently emerging field of markerless computer vision based Human-Computer-Interaction (HCI, for more details see e.g. [EBN+07]) also necessitate online processing on

behalf of the image processing system. Interaction essentially does not lend itself to offline solution: "offline interaction" is contradictory in itself. As immersive user-experience is decisive regarding the acceptance or refusal of such applications, the RT aspects of such systems are accentuated, requiring "as-fast-as-possible" implementations in order to minimize latency between user action and system reaction. As methods aiming for success should live up to the RT expectations, often the need arises for multi-threading and/or distributed processing among different computing nodes (e.g. PCs or smart cameras). Moreover, though "offline interaction" does not exist, in order to make repeatable experiments and algorithm analysis possible, implemented systems should preferably also be able to process previously recorded inputs.

1.4 Implementation issues

As both offline monocular and multiview systems can be treated as subsets of online multiview systems, such a system can also be used for offline processing. Creating an application framework, which is flexible enough to easily adapt to changes in the number of cameras, threads and computing nodes, is a formidable software engineering challenge. During research phase, work is being focused on algorithm development and quickly creating a working prototype. As scalability and adaptability tend to be secondary issues, they can be traded for RT performance, resulting in research prototypes that are usually custom made solutions to given imaging hardware and node distribution. This hinders future research, because experimenting with different algorithms or distribution models requires the time-consuming rewrite of significant components of the system. The components that have to be reimplemented or modified are responsible for interfacing with the used camera device drivers, thread management, synchronization and network communication. Also, if a totally new system is created from scratch, such components have to be written once again or previous ones have to be customized for the new system setup.

1.5 libugrab

If the system components mentioned above were available and could easily be combined programatically with each other to form a basis for complex solutions, research and prototyping could solely focus on high-level issues like algorithm selection and development, instead of also committing resources to low-level implementation aspects that are scientifically irrelevant, but must be taken care of.

This fact was our main motivation to develop the 'libugrab' C++ grabbing framework. The library can be used as a basis for creating online multiview systems. The framework was designed with flexibility in mind in order to provide easy ways to carry out the following tasks that frequently occur during vision system development:

- adding or removing computing nodes to/from the system (in order to cope with computational complexity via load-balancing – processing power is inexpensive),
- change of camera types (e.g. camcorder to DCAM),
- adding or removing of cameras to/from the nodes (off-the-shelf imaging HW is inexpensive),
- separating low-level image preprocessing into background threads from the high-level processing thread (in order to better exploit multi-core CPUs)
- notifying the main processing thread of data-arrival after preprocessing via events (without the need for polling),
- changing preprocessing algorithm (e.g. color segmentation to background subtraction),
- conversion of raw camera data to low-level processing input type (e.g. bayer to RGB),
- recording from multiple sources,
- changing from camera input to prerecorded data,
- sensor fusion.

The library handles widely used imaging sources (e.g. camcorders or webcams) and videoformats, as well as several format conversions out-of-the-box. Moreover, the user can register her own custom sources/converters, which can be used by the system like its built-in components. Extra care has been taken to make the library "unintrusive": to minimize the amount of user code required to take full advantage of 'libugrab', allowing researchers to concentrate on algorithm design and implementation.

2 RELATED WORK

There are an abundance of optical motion capture systems commercially available (e.g. [Sys07]), along with marker-based tracking products for HCI (e.g. [Gmb07]). These systems offer turnkey solutions for the specific task by realizing online multiview systems. They have a modular design, therefore the observed area or accuracy can easily be enlarged simply by adding more cameras and possibly processing nodes to the system. Their drawback from our point of view is that they are closed systems designed specifically to provide a RT solution for the visual marker-based tracking problem, often coupled with near-infrared imaging. Although e.g. [Sys07] has products to

capture multi-camera streams for offline processing, the camera images are not available online to the user, thus, it is not possible to experiment with custom processing algorithm chains. Therefore, these systems are not capable of realizing general multi-camera online systems. Another downside is that although low cost high-performance computing power and off-the-shelf imaging hardware becomes more and more available, commercial solutions tend to utilize their own proprietary HW.

'OpenCV' [Lib07] provides portable functionality to capture from different video sources connected to one computer. This grabbing interface, however, supports only pull-access (polling the sources) and does not directly makes multithreaded background or network transparent processing possible. 'unicap' [uni07] is an extensible C-library for UNIX-like OS-s providing a uniform interface for various imaging devices. Although it is possible to register a callback for arriving data, which is called in the background, the user has to take care of thread synchronization and data passing between the background and main processing threads or any kind of network transparent operation.

Multimedia Frameworks (MMF, e.g. GStreamer [GSt07] or DirectShow [Dir07], among numerous others) are media-streaming architectures, designed to handle media on a computer, usually also in a network transparent way. Multimedia data passes through a so called rendering- or filter-graph through filter nodes from the source to the sink. During e.g. playback, the source is a movie file that will be demultiplexed and decoded to video and audio data in the filters and finally rendered in the video and audio sinks. A similar setup can realize also to grab from an image source to the harddisk. Network transparency is achieved by sources that can read media data from the network and sinks that can write to the network. The 'NMM' presented in [Loh05] allows even more advanced cross-network operation, enabling the nodes of the filter-graph to transparently be instantiated on different computing nodes.

MMFs realize graph dataflow processing in separate threads and an arbitrary number of custom filter nodes can be inserted along the processing path, thus general vision algorithms can be implemented. As MMFs were designed to record or playback from a single source, achieving such goals programatically can be solved only with a few lines of code. Building a custom processing graph with proper error checking from scratch, however, requires significantly more coding effort. It is even more problematic to have more than one source in the graph (which is, however, required for multiview systems), because synchronization of different source paths with each other and possibly with the main thread has to be taken care of manually within the given framework. Furthermore, as the main purpose of MMFs is to "sink" the data to the screen (or harddisk), getting the processed data out of the graph can also be problematic. These facts, unfortunately, mean that low-level software aspects of algorithm implementation will have increased importance and have to be mixed up with "pure" algorithm implementation – [Wim05], for example, deals with the problem of implementing a stereoscopic player and multiplexer within Microsoft's DirectShow framework.

MMFs can be considered as a subset of Synchronous Data Flow (SDF) systems, with a specialized scope for multimedia processing. SDF is a special case of data flow in which the number of data samples processed by each node (filter) on each invocation is specified a priori [LM87]. SDF graph programming environments are widely used for DSP and FPGA programming and special extensions have been proposed in [SK02] for problems encountered in RT vision. SAI [Fra04] enhances data flow stream processing into a hybrid (shared memory and message passing) design framework for distributed asynchronous parallel computation targeted to realize "Immersipresence" applications. Example vision applications using the SAI paradigm clearly demonstrate that image processing enormously benefits from parallel processing and that multithreading is not an option, but a "must have". The main difference between 'libugrab' and the SAI programming paradigm is that while SAI is a design methodology for whole applications, based on a data flow-like code execution model, 'libugrab' serves as fast algorithm prototyping tool retaining scalability at the code "near" to the imaging hardware.

Finally, distributed system frameworks (DSFs) should also be mentioned. The purpose of DSFs is to make it possible to implement algorithms or applications distributed on a network, utilizing resources wherever they are available. Resources in this context are interpreted in a broad sense, meaning not only HW objects, but software entities implemented on a specific network node, too. There are a large number of such frameworks available and it is impossible to review even a fair portion of them here, for an overview over DSF programming models and middleware we refer to [BCP07]. CORBA [COR07], for example, is an object-oriented middleware and lets programmers instantiate and use objects in a network transparent way. Such general frameworks can be used to implement e.g. MMFs; this is the case with DirectShow, which is based on COM [Tec07].

3 DESIGN RATIONALE

The data flow paradigm can naturally be used to describe stream-oriented image processing systems, therefore it forms the basis for 'libugrab'. In order to allow the construction of the grabbing graph with minimal programmer interaction, we have to limit the

possible data flow graph structure. In the following we derive the graph configurations allowed by the system, that are complex enough to model the vision pipeline of a large number of classes of online multiview systems and at the same time simple enough to facilitate automatic graph construction.

3.1 An effective multithreaded application setup

Let us consider a simple monocular handtracking system that could be used as a simple alternative mouse-like input device. The input is read from a DV-camcorder, which observes the user's hand from above. Low-level processing consists of decompressing the DV frames to grayscale images and subtracting the background in order to get a binary mask of the user's hand. A high-level processing step follows that determines the 2D-position and gesture of the hand based on the hand-mask. A naïve sequential implementation of such a system is depicted in Figure 1.

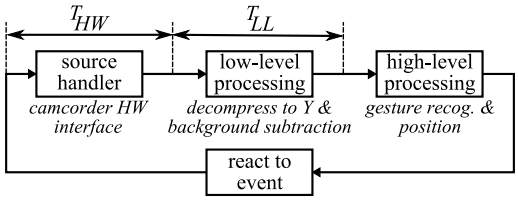

Figure 1: A naïve sequential main loop implementation of a simple mouse-like visual interface. T_{HW} is the time needed to get the DV-frame from the camcorder to main memory and T_{LL} is the duration of the low-level processing.

There are two problems with such an implementation. First, one theoretically has to wait T_{HW} time for the next raw frame (wait until the start of the next full frame and then wait until that frame arrives). Second, T_{LL} delay happens before high-level processing can take place, as the raw frame has to be converted to some appropriate format for the actual low-level processing and then the low-level processing still has to be carried out. This results in at least $T_{HW} + T_{LL}$ lag for the user. Fortunately, in the case of push-sources (like camcorders or DCAMs in isochronous mode) T_{HW} can be neglected, as the OS device drivers adapt to the nature of the imaging source and buffer the raw frames. T_{LL}, however, remains, though clearly could be drastically reduced or even eliminated on modern multi-core systems, simply by doing the low-level processing in a second thread.

A subtle additional problem is that if we would like to update our example system to multiview, e.g. to triangulate the position of both hands to allow 3D interaction, the individual $T_{HW} + T_{LL}$ lags are additive, see Figure 2.

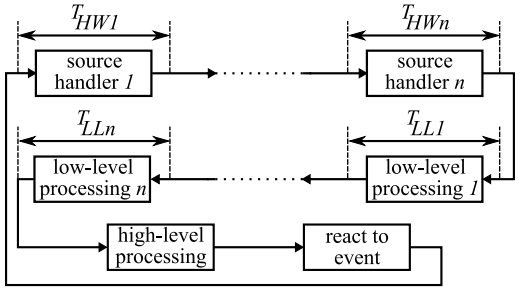

Figure 2: In the case of a multiview system the $T_{HW} + T_{LL}$ individual lags are additive.

Naturally, the solution to these problems is to create a separate grabbing thread for each source. These grabbing threads should carry out the low-level processing and store their results in buffers. The high-level algorithm runs in its separate thread and accesses the preprocessed data in a thread safe way, c.f. Figure 3. This setup has the extra benefit that it fits the operation mode of push-sources and does low-level processing on demand as the data arrives. It is clear that efficient multiview systems all have such a setup, thus the creation of such systems should be possible applying the same pattern.

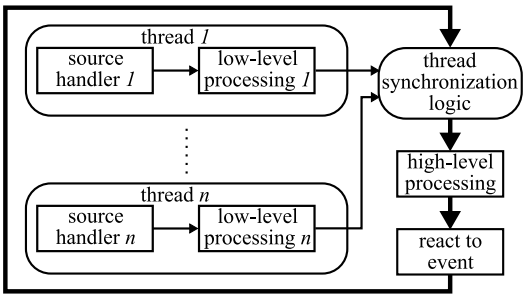

Figure 3: Efficient multithreaded setup of a vision system. The thick arrows represent the high-level thread that takes input data from the low-level threads.

Vision systems should solve the given computer vision problem they were designed for. Implementing a multithreaded setup for an online system plays an important role in this task. In the following we will examine what parts of the full solution should come necessarily from the problem domain addressed by the system and what tasks are common enough to be abstracted away into a library that allows for the automation of setting up the systems described above.

3.2 Changes in the low-level thread

Let us examine what happens if we change different algorithmic aspects of a low-level processing thread. We will do this by modifying the initial setup of the 2D mouse-like interface example described previously.

First, suppose that the camcorder is changed to a bayer format DCAM, in order to gain higher resolution and framerate. This change does not have any influence on the grayscale background subtraction step. The camcorder and the DCAM, however, have fundamentally different raw frame formats, thus the "raw-to-algorithm-input" conversion has to be reimplemented. On the other hand, if we change the background subtraction to skin-color segmentation in HSV-space, both the "raw-to-algorithm-input" and the actual mask-generation algorithm have to be recoded. These problematic algorithm parts are illustrated in Figure 4.

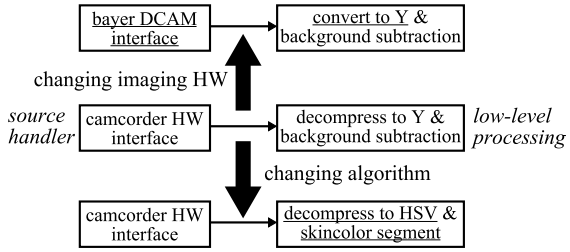

Figure 4: Modifying the low-level thread. Changes in the source and the processing algorithm lead to different reimplementation issues; reimplementing the conversion code, however, is necessary in both cases.

It follows from the above facts that conversion from raw source data to the appropriate low-level algorithm input can be implemented independently both from the type of the source and the actual low-level algorithm. As long as the source and algorithm use a common specification (that of the library) to specify their output and input, conversion code can be automatically selected and provided by the library. Moreover, interfacing to different sources is also independent from the processing code, therefore code to handle different sources should also be reused.

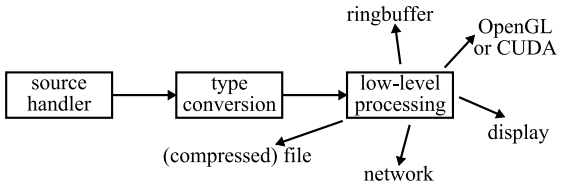

Figure 5: Different uses of the results of the low-level thread. They are all independent of actually what kind of processing was carried out in the low-level box.

As shown in Figure 5, there are several possible ways to further process the results of the 3 stages of the low-level thread. It should be noted, however, that if the size (width, height, padding) and the endianness of the result are known, any of these additional processing possibilities can be abstracted away by a "sink" (it is not exactly true for OpenGL sinks, but if the types are assumed to be known a priori before operation, this problem can be taken care of during sink initialization). Thus, different sink functionality can also be accumulated in the library and reused.

In accordance with the above discussion the final low-level thread can be decomposed into 4 steps as illustrated in Figure 6. This decomposition scheme has several benefits:

- because of the fixed size of the graph nodes, such a graph can be created automatically by a framework if the source, low-level algorithm and the sink are given,
- as sources and sinks are automatically provided, the relevant work can focus in implementing the low-level algorithm,
- such a framework does not necessarily have to be used with imaging sources and thus, for image processing.

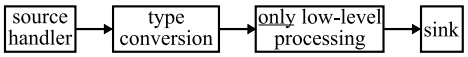

Figure 6: Final decomposition of the low-level thread. All but the low-level box should be provided by the library.

An important aspect not addressed by the scheme of the several 4-stage low-level and one high-level threads is data multiplexing. Disparity maps, for example, could be computed in the background from two images rectified in low-level threads, before actually reaching high-level processing. As in the design discussed so far the low-level threads are independent from each other, this is not possible. To alleviate this problem, multiplexers (see Figure 7) should also be provided by the library. As multiplexers have sinks, they can be cascaded arbitrarily.

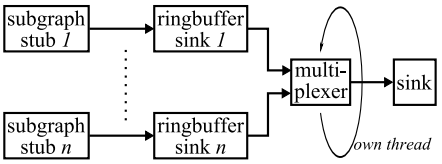

Figure 7: Multiplexing several low-level stubs with ringbuffer sinks (stubs are the subgraph composed of the source, converter and low-level processor nodes of the low-level thread).

3.3 Rationale summary

'libugrab' uses a two-stage model for per node processing: there is a low-level and a high-level stage. Low-level processing is driven by the capturing device running in its own thread. Any algorithm that is used in

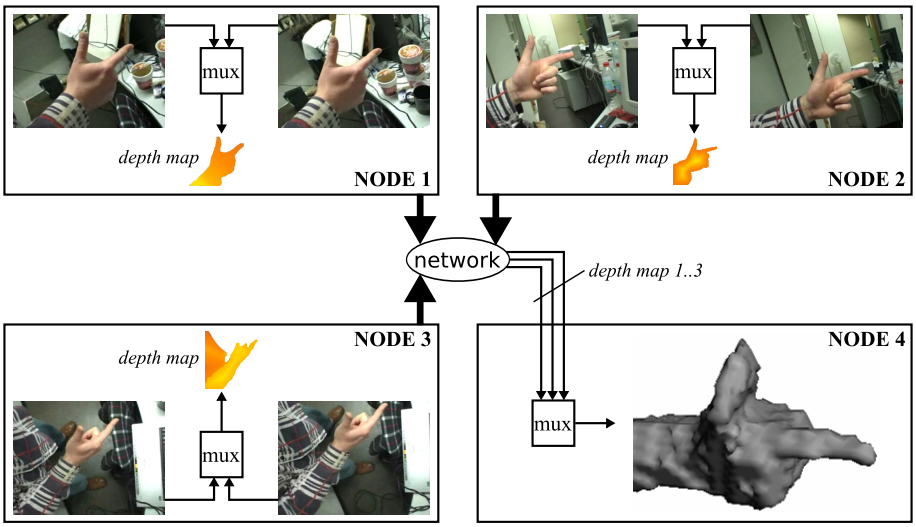

Figure 8: A relatively complex 3D reconstruction system easily realizable by ‘libugrab’.

the low-level stage is assumed to be fast enough to be executed between successive frames. The high-level stage depends on one or more low-level preprocessing results and is carried out when necessary data arrives. Low-level output can be used directly by the high-level stage or multiplexers can be employed as an additional intermediate step to combine several low-level results for high-level processing. The multiplexing and the high-level step are detached from the low-level threads through the use of buffers, consequently there are no assumptions made about its execution time. Network-transparency is achieved similarly to MMFs by allowing (possibly multiplexer) sinks to write to the network and sources to read from the network. Data streaming directly supported by ‘libugrab’ is downstream: from the imaging sources to final high-level processing thread, where some relevant result should be computed. An example for a relatively complex system that is easily realizable is depicted in Figure 8.

4 IMPLEMENTATION DETAILS

In the following we describe some parts of the library to provide better insight into its internal workings.

4.1 Type system

Types build the fundamentals for automatic graph building. Determining whether two graph nodes can be connected is based on checking their output and input types. Types are stored in the type registry, in distinct “type-trees”. Type-trees are a hierarchical collection of connected types. Two types can be equivalent, have a subtype relationship between them or no relationship at all. Leaf elements in the type-trees are actual types, other elements are general types. In a created graph all input and output types should be leaf-types. In order to allow users to implement their own sources with output

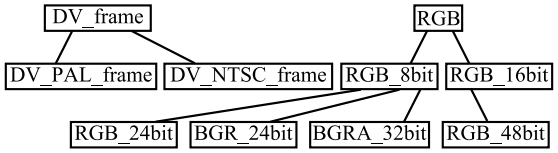

Figure 9: Example of “type-trees” in the type registry.

types not included in the library, custom types can be added to the registry. The relationship of some built-in types is shown in Figure 9.

4.2 Automatic graph building

Maybe the most important feature of ‘libugrab’ is the automatic graph-building. In contrast to other systems, the structure of realizable graphs is restricted, this, however allows automatic creation of low-level subgraphs with one function call. A graph can hold any number of subgraphs, the multiplexers with their sinks count as subgraphs in this regard, too.

In order to create a subgraph (without multiplexer output) the user must specify the source, the low-level processing algorithm and the sink. Sources are specified by string, for example ‘PUSH1394_00097eff51200083’ refers to our firewire bayer DCAM with GUID ‘00097eff51200083’ and ‘/dev/v4l/video0’ to a connected webcam. This means that by changing the source string literal (possibly by taking it from the command-line) one can easily use different sources for processing.

The essence of the subgraph is the processing algorithm. This is the only important piece that the user has to provide. It can simply be done by inheriting from the `Callback` or `Inplace_callback` classes. The callback should specify its input type along

with the desired storage properties: alignment of the input data pointer and row padding. These two parameters are needed, since special processing implementations, e.g. using SSE2 instructions cannot work with arbitrary alignment and padding. When there is no need for callback processing in the subgraph, the user should simply specify needed type and storage properties.

Finally, a sink must be specified. Either the user instantiates one implemented in 'libugrab' or can sublass `Memory_sink` or `Nonmemory_sink` to provide her own. These two types of sinks are distinguished in order to allow "optimal" subgraph building without extra buffering between the nodes. A memory sink (e.g. a ringbuffer) provides its own data area where it requires the result of processing to be placed, in contrast to e.g. a sink that can flush its input to network on the fly.

The library maintains a registry of source factories, which are all queried whether they can create an instance based on the source ID string. If one is found, it is used to create the source. The source gets the required processing input as a hint. If possible, it can obey this request, however, it does not have to. If the source does not provide the requested type, a converter is created with the help of the converter factories. The user can register her own sources or converters that will be automatically picked up by the subgraph creation process. Finally, the callback (if any) and the sink are initialized and the subgraph is ready for processing. Subgraphs connected to a multiplexer can be similarly created by delayed initialization of the subgraph stubs. For example, to create all the low-level graphs on 'NODE 4' in Figure 8 the user has to make 4 function calls: 3 times adding the appropriate stubs with the network sources and finally adding the multiplexer with its sink.

4.3 Getting data out of the graph

Until now, we have created all the subgraph processing threads. The main thread depending on the background threads still has to be notified about the fact that data has arrived for processing. In order to allow this, the subgraph nodes can specify signals (based on condition variables to avoid missed signals), which can be subscribed to by another (usually the main) thread. Using these signals and a special helper class (the `Signal_multiplexer`) the thread in question can wait for any, a subset or for all subthreads to produce data. In order to avoid the starvation of slower sources, the `Signal_multiplexer` checks for arrived signals in a Round Robin fashion. Please note, that this signaling scheme allows for not only waiting for new data, but any "interesting" event: e.g. the end of the learning phase of background subtraction or framedrop.

There is a special signal, the graph error signal, which must be subscribed to if any other signal of the graph is subscribed to. This is enforced in order avoid deadlocks if the thread listened to does not produce data any more for some reason.

4.4 Error handling

As mentioned above, every graph has an error signal. It must be signaled by any subgraphs of the graph if an error occurs. This way, the graph can stop all of its subgraphs in an exceptional case. If a graph is spanning multiple computing nodes, the error condition is propagated through the network on special "service" sockets created by the involved network sinks and sources.

4.5 Pull sources – offline processing

As previously argued, it is very useful for algorithm debugging or evaluation purposes to be able to process prerecorded input. The user can specify video-files, directories with images, etc. as sources for this purpose – simply by changing the source ID string. In this case, a special two-threaded pull-source is created that pumps data into the graph. One thread reads the next data chunk, the other pushes the data down the subgraph. Framedrop is avoided by suspending the reader thread until the sink of the subgraph finishes processing.

5 EXAMPLES

The first example application of the system was to implement multi-node/multi-source grabbing based on 'libugrab'. Actually the main implementation effort was to create a special writer multiplexer. This special writer multiplexer allocates as large ringbuffers for the local sources as possible and simply writes the arriving frames with their timestamps (timestamps are created in the sources for the data coming from the devices) into a common file. After recording, frames with the smallest time difference are merged across the network from the common files into streams that have the same number of frames. This was implemented as an extra function (that is why we have created an extra application for it), could have been done theoretically in the sink destructors, though. In order for this simple scheme to work, we had to synchronize the clocks on our local network. This has been done by 'ntpd', as suggested in [Loh05].

We used our library to implement both the 2D and 3D interaction systems mentioned in the Design Rationale with near-infrared background subtraction and skincolor segmentation, too. 'libugrab' was also used to create a 3-view model-based handtracking system based on adaptive color segmentation. The segmentation was implemented as a callback and the skincolor distribution was updated based on the registered hand-model area in the camera images. As the model based registration was not as fast the grabbing sources, this update had to be asynchronously injected into the subgraph threads. Though 'libugrab' does not support upstream data out-of-the-box, the update was safely real-

ized by using mutexes for the distribution access in the implemented callback.

We have found the greatest use of the library during the development of our RT markerless handtracking system ([SKSK07]). The first prototype setup consisted of 3 cameras with preprocessing on 3 Linux PCs and a master Windows PC carrying out the high-level algorithm. Later, when we optimized both low- and the high-level processing, we were able to migrate the system seamlessly to a single Windows computer thanks to 'libugrab'.

Finally, we also used the library to implement a toy surveillance system that consisted of 5 PCs in 5 rooms and a master PC. Background subtraction was used to detect whether someone was present in any of the rooms. The slave PCs sent just a boolean value to master, simulating programmable TCP cameras.

6 CONCLUSIONS AND FUTURE WORK

We introduced the design and implementation of 'libugrab', a versatile grabbing library. The main design goals were efficiency and to allow researches to concentrate on the scientifically significant parts of algorithm development. We demonstrated with several examples the ease-of-use of our library.

We plan to release the full library under BSD-license in the near future (apart from the code that uses GPL/LGPL licensing). In order to be able to do this, the most important task is to create better documentation. Releasing the library would also allow for anyone to participate in the development effort and contribute new ideas. Please visit our website at 'http://cg.cs.uni-bonn.de/project-pages/libugrab/'.

We have several ideas to improve the current implementation, like driver programs configurable by plugins for simpler graphs and to allow subimage processing as data is streamed from the cameras, as opposed to the current frame oriented implementation.

REFERENCES

[BCP07] Alex Buchmann, Geoff Coulson, and Nikos Parlavantzas. http://dsonline.computer.org/middleware/, December 2007.

[COR07] CORBA. http://www.omg.org/, December 2007.

[Dir07] DirectShow. http://msdn2.microsoft.com/en-us/library/ms783323.aspx, December 2007.

[EBN+07] A. Erol, G. Bebis, M. Nicolescu, R.D. Boyle, and X. Twombly. Vision-based hand pose estimation: A review. 108(1-2):52–73, October 2007.

[Fra04] Alexandre R.J. François. A hybrid architectural style for distributed parallel processing of generic data streams. In *Proceedings of the International Conference on Software Engineering*, Edinburgh, Scotland, UK, May 2004.

[Gmb07] Advanced Realtime Tracking GmbH. http://www.ar-tracking.de, December 2007.

[GSt07] GStreamer. http://gstreamer.freedesktop.org, December 2007.

[Lib07] Open Source Computer Vision Library. http://www.intel.com/technology/computing/opencv, December 2007.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE, Vol. 75., No. 9*, September 1987.

[Loh05] Marco Lohse. *Network-Integrated Multimedia Middleware, Services, and Applications*. PhD thesis, Department of Computer Science, Saarland University, Germany, June 2005.

[SK02] Dirk Stichling and Bernd Kleinjohann. CV-SDF - a model for real-time computer vision applications. In *WACV 2002: IEEE Workshop on Applications of Computer Vision*, Orlando, FL, USA, December 2002.

[SKSK07] M. Schlattmann, F. Kahlesz, R. Sarlette, and R. Klein. Markerless 4 gestures 6 dof real-time visual tracking of the human hand with automatic initialization. *Computer Graphics Forum*, 26(3):467–476, September 2007.

[Sys07] VICON Motion Systems. http://www.vicon.com, December 2007.

[Tec07] Component Object Model Technologies. http://www.microsoft.com/com/, December 2007.

[uni07] unicap. http://unicap-imaging.org, December 2007.

[Wim05] Peter Wimmer. Stereoscopic player and stereoscopic multiplexer: a computer-based system for stereoscopic video playback and recording. In *Stereoscopic Displays and Virtual Reality Systems XII, Proc. of SPIE Vol. 5664A*, pages 400–411, 2005.