EGVE Symposium (2008)
B. Mohler and R. van Liere (Editors)

BAT - a distributed meta-tracking system

Ferenc Kahlesz and Reinhard Klein

email: {fecu,rk}@cs.uni-bonn.de

University of Bonn
Institute of Computer Science II
Computer Graphics Group
Germany

Abstract

This paper describes the design of the 'BAT' (Bonn Articulated Tracker) visual tracking framework. This system allows the easy implementation of real-time, multi-camera motion tracking that can be distributed (also in multi-threaded sense) across several computing nodes (or CPU cores). The system in itself does not realize any specific tracking system, but manages a meta-algorithm flow between processing blocks. An actual tracking implementation is realized by specifying the processing blocks through plugins. Depending on the plugins supplied, 'BAT' is capable to instantiate a wide-variety of systems ranging from object-detection methods to model-based deformable object tracking based on time-coherence, allowing also for hybrid algorithms. Being a "meta dataflow system", 'BAT' also naturally facilitates sensor fusion. Moreover, it can be used as a testbed to compare and evaluate different kind of tracking algorithms or algorithm substeps.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Input devices, Parallel processing

1. Introduction

Examining the area of 'vision-based motion tracking' one can find a multitude of different kinds of algorithms: direct object-detection methods, model or feature based tracking systems exploiting time coherence and hybrid-systems fusioning the previous two disciplines. Also the goals existing methods solve differ strongly: how many objects should be tracked, whether these objects are two or three dimensional and/or rigid or deformable, the class of 2D/3D transformations these objects may undergo, whether the systems employ monocular or multi-camera imaging and whether they aim real-time or offline processing.

1.1. Tracking for Human-Computer-Interaction

The development of markerless visual tracking systems for Human Computer Interaction (HCI) in Augmented or Virtual Reality (AR/VR) environments is an especially demanding task. The aim of such systems is to replace conventional tracking methods like datagloves or electro-magnetic position and orientation tracking devices, in order to allow more instantaneous, natural and immersive user experience by not requiring the user to put on, possibly calibrate and wear any extra devices during interaction. Such systems would enormously benefit AR/VR applications *e.g.* at exhibition booths or in virtual museums, where it would grant access to the AR/VR content for a much larger number of users from the audience. Currently, available technology in the field of markerless visual tracking limits the realizable AR/VR systems that rely only on such methods as input devices, despite the enormous research activity carried out in the fields of markerless body [Gav99] [MHK06] and hand tracking [EBN*07] [dC06] since the beginning of the 1990s.

The main difficulty of realizing interface quality tracking systems is that interaction does not lend itself to offline solutions: "offline interaction" does not exist. Offline tracking algorithms, however sophisticated and stable, cannot be applied in practice. Moreover, as immersive user-experience is decisive regarding the acceptance or refusal of the tracking system as an input device, the system should even track the users' actions "as-fast-as-possible" in order to minimize la-

© The Eurographics Association 2008.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

tency between user action and the reaction of the AR/VR system.

Another important problem is that lost tracking detection and (re)bootstrapping should preferably happen without any user assistance in order not to hinder seamless interaction.

1.2. Visual HCI and multiple-target tracking

Though state of the art literature of visual HCI focuses on tracking the body of one person or her hand in particular, general HCI input systems should track multiple objects, *e.g.* multiple persons for telepresence or both hands for two handed object manipulation. Thus, tracking for visual HCI can be considered as a special case of the broader scope of general purpose tracking systems for multiple, possibly articulated/deformable targets in real-time. Besides visual HCI, such systems can serve numerous other purposes *e.g.* security video surveillance, automatic sports video analysis or traffic monitoring.

Therefore, although the idea of 'BAT' originates from the visual HCI field and we use examples from this application domain to demonstrate the capabilities of our system, our concept benefits the more general field of multiple-target tracking.

1.3. Computational complexity

Due to the curse of dimensionality, algorithms used in visual HCI tracking tend to be computationally expensive, like non-linear model-fitting to image evidence or maintaining distributions in particle filters. Computational complexity can be challenged with any combination of the following approaches: (1) developing computationally less demanding algorithms (2) using heuristics, (3) using special purpose HW for algorithm (sub)steps and (4) distribute the implementation among different computers or processor-cores via multithreading or both.

In this paper we concentrate on the last approach: distributed implementation and multi-threading. We consider this an important problem, because given the state-of-the art in high-DOF articulated object tracking algorithms and off-the-shelf HW *it is the only way to achieve usable framerates for visual HCI* and thus, to reduce the overall end-to-end latency of AR/VR systems to a level where usability studies about interaction metaphors can be conducted (if the only bottleneck is the tracking latency).

Besides being such an enabling technology, having an easy way to implement distributed tracking has other advantages. If the mathematical algorithm used supports parallelism, this can be straightforwardly exploited. Increasing robustness or tracking volume is also facilitated by simply adding new cameras to the system, even if it necessitates the presence of new computers.

Please note, that our focus is the distributed implementation of *centralized* tracking systems. Decentralized tracking methods and distributed sensor networks [CES04] are a distinct topic.

1.4. Distributed implementation issues

Implementing distributed tracking systems requires the development of infrastructure-level code for multithreading and/or network communication and appropriate synchronization logic. This can be done bottom-up using OS facilities or top-down, by using some distributed middleware. The amount of infrastructure-code in both cases can easily outweigh the amount of algorithm-level implementation.

Interesting parts of creating a tracking system from a research point-of-view are the mathematical development of an algorithm and its implementation. If there is some "hard limit" on execution time, which requires distributed implementation (like in visual HCI), scalability and adaptability are usually traded for performance. Thus, research prototypes tend to be custom solutions where algorithm implementation is intertwined with low level infrastructure code for a specific communication topology between computers and/or threads.

Because of this, making changes to the realized algorithms or algorithm execution distribution requires the time-consuming rewrite of significant components of the system, partly infrastructure-level, partly algorithm-level. This clearly hinders research.

Therefore research-level code must be separated from infrastructure-level code. Given such separation, research and prototyping could solely focus on high-level issues, instead of also committing resources to software infrastructure.

1.5. BAT

'BAT' provides the mentioned separation for the case of distributed multicamera, multi-object tracking. After developing several prototype distributed hand-tracking systems, a pattern has emerged as to what parts of the systems belong to algorithm-level and what to infrastructure-level implementation. The infrastructure-code has been formalized as 'BAT'; its core constitutes a possibly network and/or thread transparent meta-algorithm-flow (MAF) implemented in C++.

The MAF describes the dataflow of a general tracking system that can be specialized to instantiate a concrete algorithm by specifying dataflow processing nodes through plugins. Specifying the plugins also determines the type of the data flowing across nodes. The name of the plugins to use along with their network execution place are defined in a configuration file.

Figure 1 illustrates the metaflow concept for the simple

© The Eurographics Association 2008.

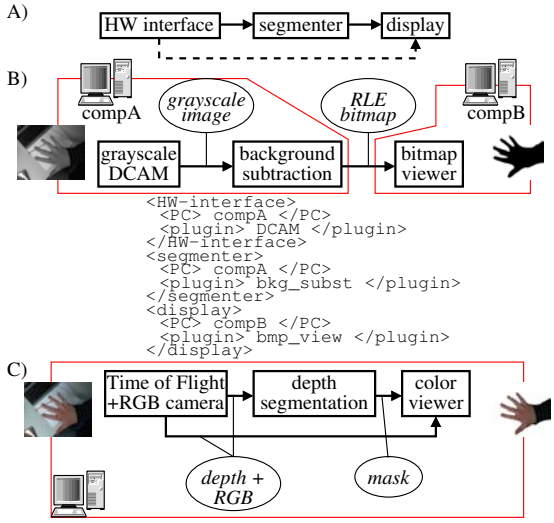

Figure 1: *The meta-algorithm-flow (MAF) concept. A) MAF of a segmenter algorithm. The rectangles are the plugin placeholders. The dashed line depicts an allowed, but not required connection. B) An instance of the segmenter algorithm distributed on two computers using DCAM input. The text in the ellipses shows the data-types for this instance. (RLE: run-length-encoded). Below the image a theoretical XML configuration file for this instance. C) Instance on a single computer with TOF/RGB input. The possible dashed connection from A) is also utilized for color output.*

case of a theoretical segmentation algorithm (this is *not* the MAF of the system, which will be described later in Section 3.3). Figure 1 B) also demonstrates how the functionality and distribution can be specified in a system configuration file. The input and output datatypes of the plugins must match each other for successful operation. As the plugins must define their in/out types, this can be checked by the system on startup.

As 'BAT' takes care of network communication and multithreading, only algorithm-level code has to be implemented through the plugin interfaces. Finally, it is important to note what 'BAT' does *not* do:

- It does not schedule the network or CPU-core distribution of the plugins for optimal performance. The responsibility to achieve this lies by the user and is made possible through the system configuration file.
- 'BAT' is not a general dataflow system. Its MAF is geared toward tracking and it is not possible to add flow-processing nodes outside the meta-flow.

2. Previous work

Several papers have proposed frameworks for acquiring images from different camera sources on a network. Some of them [LZT06] [AIM04] concentrated exclusively on synchronized grabbing and storing data for subsequential offline processing. Others [TBAR05] [DSVG03] [TLMS03] advocated online operation. They, however, either do not specify how such processing should happen, or describe exact algorithms to be carried out. In contrast, 'BAT' specifies a framework geared toward tracking and can be seen as a tool to realize various algorithms online, without enforcing any specific algorithm.

The open-source OpenTracker [RS01] and VRPN [RMTHS*01] tracking data communication frameworks ease the implementation of distributed AR/VR systems by providing a network transparent access to different tracking devices. 'BAT' can be seen complementary to these systems, because it allows the creation tracking devices that can be used as tracking sources in them.

Another open-source project, FlowVR [AGL*04], is a middleware dedicated to cluster- or grid-based VR applications. Its purpose is to make it simpler to create a *whole* VR system, like the GrImage platform [AFM*06], by realizing an easy-to-use, *general* purpose dataflow system. 'BAT' is not that general, but aims to be much easier for the *specific* purpose of distributed tracking system implementations.

[dC06] describes a system where different autonomous trackers for different objects communicate their results to a standalone viewer via network. Apart from sending the tracked states to the viewer, the trackers operate independently. 'SAI' [Fra04] proposes a C++ framework for developing distributed, asynchronous parallel data-driven applications, including real-time vision software. 'YARP' [MFN06] is a collection of libraries written in C++ for development of distributed control software for robotics. 'YARP' seems to be born out of the similar reasons as 'BAT': (1) as stated in [MFN06] "one processor is never enough" and (2) to clearly separate "infrastructure-level" code from "research-level" code in order to allow reusability.

Both 'SAI' and 'YARP' are freely available for download. Unfortunately, they lack features for our purposes: they themselves do not make it possible to create a network transparent dataflow graph given a desired configuration.

3. System components

The system consists of four main components: a database, sensor handlers, a bootstrapper/lost tracking detector and a tracker (see Figure 2). The *database* holds the tracked states of the objects and any kind of algorithm parameters. As 'BAT' runs distributed on several computers, the database is shared on the network. The *sensor handlers* manage sensor HW.

The basic idea is that the *bootstrapper* detects objects and verifies the validity of the currently tracked states in a very robust manner, but is not fast enough itself to provide the

© The Eurographics Association 2008.

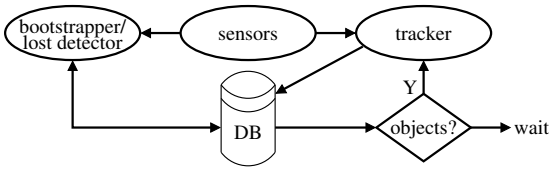

Figure 2: *Overview of the main components and their communication.*

required tracking framerate. It introduces new objects to the database and deletes lost objects.

The *tracker* runs only if there are objects present in the system. The role of the tracker is to maintain a usable state update frequency during the execution time of the bootstrapper, most probably based on time coherence of the tracked states. If the tracker loses tracking of some objects, this fact will be noticed either by itself or the bootstrapper and the objects in question will be deleted from the database. If the bootstrapper is efficient enough on its own, the tracker is not needed. This is the special case of a purely detection based system.

In the following we will describe the main building blocks of the meta-system. The most interesting is the metaflow of the bootstrapper (the tracker has identical MAF), which is described in Section 3.3. For examples of how concrete choice of the plugins and their connection can realize actual tracking systems, please refer to Section 4.

Although the dataflow will be considered in a networked environment, 'BAT' can instantiate a full-fledged tracker also on a single computer using multithreading.

3.1. Network database

The network database serves two main purposes. First and foremost, it holds the states of currently tracked objects. These states are synchronized after every tracking cycle over the network. Second, it stores system-wide parameters for the different plugins. These parameters can be any kind of data interesting in an actual tracking system, *e.g.* skin-color distribution for a color segmenter, thresholding values or CAD model descriptions of object classes for a model-based tracker, etc. Such records in the database are synchronized only if they were modified and are sent only to the computers where they are referenced from the plugins. As the database can be accessed and written from an external program, this allows online tuning of algorithm parameters or visualization of the tracking results.

Database instances on a computer can also hold local records that can be used for out-of-band communication between not-directly-consecutive dataflow nodes (plugins) residing on the same computer. If $A \rightarrow B \rightarrow C$ are three successive nodes of the dataflow and A and C exist on the same PC then A can pass on data to C via the local records directly without the need to send it through B with networking involved.

3.2. Sensors

The input sensors are handled by user supplied plugins that continuously inject measurements into the input ringbuffers of the 'BAT'. They run in their own threads, which allows measurement preprocessing (*e.g.* background subtraction or filtering) in the background, if needed. Although usually the boostrapper and the tracker share the same set of input sensors, they are allowed to use different measurements sources.

By implementing sensor plugins that present the next measurement only if their output have been consumed by all dataflow nodes that reference their measurement (more on this in Section 3.3), it is possible to change the input datafeed from online to offline. This facilitates algorithm evaluation using prerecorded data.

3.3. Bootstrapping and lost tracking detection

This section introduces the main bootstrapping metaflow. First, a high-level overview is given, then we describe the details of distributed execution.

3.3.1. Overview of the MAF

Figure 3 depicts the MAF of the tracking algorithms that are realizable by our system. The rectangular boxes represent plugins. Clearly, by specifying different plugins, different kind of tracking algorithms can be instantiated (examples for concrete trackers are present in Section 4). Although for the sake of clarity the algorithm flow is presented as a monolithic stream in this high level overview, multiple instances of the plugins denoted with an asterisk are allowed, *e.g.* multiple sensor handlers for multiview input. This is described in further detail in 3.3.2.

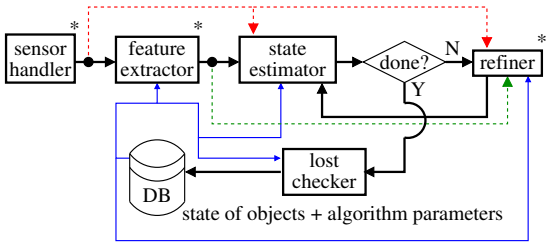

Figure 3: *Bootstrapping MAF overview. Please refer to Section 3.3.1. Asterisk above a plugin means that multiple instances of that plugin are allowed.*

The input data from a sensor (*e.g.* color camera) is pushed into the system by the *sensor handler*. The next step is carried out by the *feature extractor* (*e.g.* color segmentation or

© The Eurographics Association 2008.

edge-detection). Based on the results of the feature extraction, the state of the tracked objects is estimated by *state estimator* (*e.g.* gesture classification or crude positioning for later refinement).

In some cases the states of the objects can be inferred in one step from the feature extraction results. In other cases only initial state estimates can be computed that need to be refined in subsequent iteration. The second case is supported by the possibility to define a *refiner* (*e.g.* hypothesis iteration based on some kind of visual error measure).

After the states have been estimated, the *lost checker* verifies the states produced by the tracker based on the estimates of the *state estimator*. The exact way this is carried out is tracking system specific and depends on the number of allowable objects or object-classes. Nonetheless, the *lost checker* should somehow "cross correlate" the opinion of the bootstrapper with the belief of the tracker for the same point in time.

'BAT' supports lost tracking detection by accumulating tracking results during the execution of the bootstrapper. If both the bootstrapper and the tracker use a common (sub)set of sensors, the results of the tracker for the same input can be identified based on timestamps and retrieved from the database. If this is not the case, tracking results in the vicinity of the timestamps of the measurements for bootstrapping can still be queried. Please note that it is not necessary that the tracker produces the same kind of state estimate as the bootstrapper: it is possible *e.g.* to compare probability distributions from the *state estimator* with maximum-likelihood estimates from the tracker. Of course, the database should be written in a way that can be interpreted by the tracker. In simple cases the, the *lost checker* can be omitted and its functionality implemented in *state estimator*.

The *lost checker* is not the only plugin that is allowed to utilize the database. The blue arrows from the database to the processing plugins serve two purposes: (1) to retrieve algorithm parameters and (2) to make it possible to incorporate knowledge about the state of the system during bootstrapping *e.g.* for occlusion handling. As side-effect, it also lets algorithms implement "lost-detection" in plugins prior to *lost checker*.

The red dashed arrow from the *sensor handler* to the *state estimator* and the *refiner* illustrate that these components can subscribe to measurement data if it makes sense from an actual tracking system's point of view. The green dashed arrow indicate similar subscription possibilities as with the *sensor handler*, but between the *feature extractor* output and the *refiner*.

3.3.2. Distribution of the MAF

The detailed dataflow of the general bootstrapping algorithm is depicted in Figure 4. The input originates from S sensors, handled by the appropriate plugins.

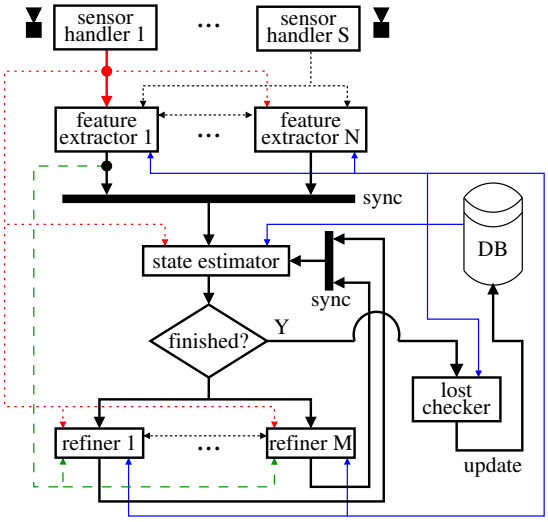

Figure 4: *Bootstrapping and lost tracking detection dataflow. The rectangular boxes represent the plugins that must be supplied by the user. The sensor and database connections to the* tracker *are omitted for clarity. See Section 3.3 for details.*

The red arrow from *sensor handler 1* to *feature extractor 1* signifies that at least one feature extractor must reference a sensor data. The red dashed arrows illustrate the subscription possibilities to the sensor handlers. For the sake of clarity such possible connections are shown only for *sensor handler 1*. If both a sensor handler and a plugin subscribed to it run on the same PC, the measurement simply remains in the main memory as long as it is needed. If not, the data will be sent asynchronously through the network. This avoids the inefficient synchronous propagation of the data through the whole dataflow.

The measurements are processed by N feature extractors ($S = N$ is not necessary). The green dashed arrows depict subscription possibilities to the feature extractors. The bidirectional arrow between *feature extractor 1* and N indicates that, if required, it is possible to allocate communication channels during system initialization between the feature extractors. The outputs of the feature extractors are fed to the *state initializer*. This is a synchronization point, because the *state initializer* is not executed until all of its inputs are available.

If the estimated states require refinement, it is handled by the M refiner plugins. Similarly to the feature extractors, the refiners can also communicate among themselves if the need arises.

© The Eurographics Association 2008.

3.4. Tracking

The tracking module has the same setup as the bootstrapper, without the *lost checker* plugin. The tracker might have its own $S_{tracker}$ sensors, $N_{tracker}$ feature extractors and $M_{tracker}$ refiners. In practice, however, it often happens that both the bootstrapper and the tracker use the same algorithm for feature extraction for mutual sensors. This can be handled as a special case to avoid executing the same algorithm for the same measurements twice. Figure 5 *B*) illustrates this for the case when the bootstrapper and tracker share all the sensors and feature extractors.

Whether this happens depends on the user: it is of course possible to instantiate the bootstrapper on different computers than the tracker, while they can still use common feature extraction. This is useful if the tracker must be in the vicinity of the sensors for performance reasons, whereas the bootstrapper can receive measurements on a slower network connection.

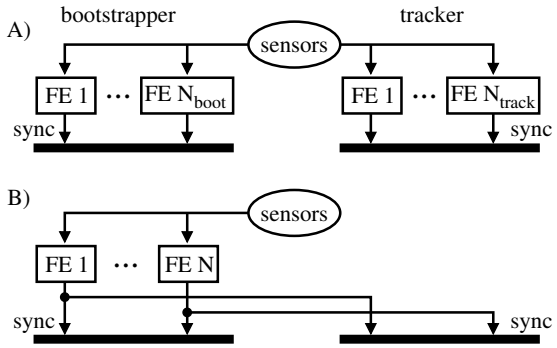

Figure 5: *Bootstrapper and tracking input with different feature extractors for both module (A) and with the same feature extractors (B). FE means 'feature extractor'.*

3.5. A note on plugins

'BAT' provides a decomposition of a tracking system through plugins and possible connections between them. The subscription facility to processing results, local (same PC) plugin communication through the database and the possibility to communicate *e.g.* the refiner plugins allows for a very flexible algorithm design.

Surely it is possible to show examples, where this kind of granularity is not enough. If a skin-color feature extractor would like to adapt its color-distribution based on the previous tracking results, it can do so only sequentially during its execution time. If this update is time-consuming, this will slow the whole tracking cycle.

Note however, that such problems can be avoided by implementing plugins that are multithreaded themselves, albeit with explicit infrastructure-level implementation provided by the user.

4. Examples

In this section we present examples of realizable tracking systems by specifying concrete algorithm choices for the different plugins and the connections between them. The importance of the user's choice about the placement of the plugins on different nodes is also illustrated.

4.1. Monocular hand-tracking on a single PC

This example system (cf. Figure 6) tracks hands with stretched fingers that appear approximately planar to the image plane. A DV-camcorder input is controlled by the *sensor handler* plugin. There is only one *feature extractor* that implements a Viola-Jones detector [VJ01] trained for hands in the *bootstrapper*. Supposing a fast enough implementation the *tracker* is not needed. The *lost checker* plugin can be omitted and the *state initializer* simply stores the detection results in the database, *e.g.* 4 float values for position, scale (≈depth) and approximate orientation per hand.

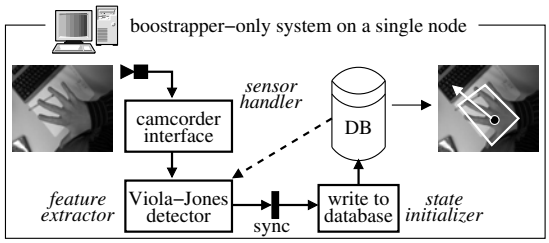

Figure 6: *Simple monocular system. The plugin names (roles) are set in italics beside the plugins. Synchronization is not needed in such a simple case, it is only shown to ease matching the system with the general dataflow depicted in Figure 4. The dashed arrow illustrates that the Viola-Jones classifier data comes from the database and serves as an example how algorithm parameters can be stored there.*

4.1.1. Modifying the system

Modifying and improving this simple system is straightforward. If one would like to use a DCAM input camera for higher-resolution or framerates, only the *sensor handler* plugin has to be replaced. By adding a *lost checker* to the system that maintains a Kalman-Filter for every hand present temporary occlusions can be handled – the extra tracked parameters *e.g.* for velocity can be stored in the database (actually in this simple case this could also be done in the *state initializer*).

4.2. Additional cameras for larger tracking volume

Suppose we would like to add additional cameras to the system described in Section 4.1 in order to track the user in a larger interaction volume. There is only certain amount of

© The Eurographics Association 2008.

cameras that can be added to the system in a single PC implementation, because either the detector will not be able to maintain its framerate for the additional images or simply the limit of allowed cameras per PC will be reached.

Solving this problem with 'BAT' is easy: the detectors can be executed on different computers. This can be achieved by modifying the *state initializer* plugin implementation (the position data must be interpreted in a camera-dependent context) and updating the system configuration file to let 'BAT' instantiate additional sensor handlers and detectors on other computers.

4.2.1. The responsibility of the user for distribution

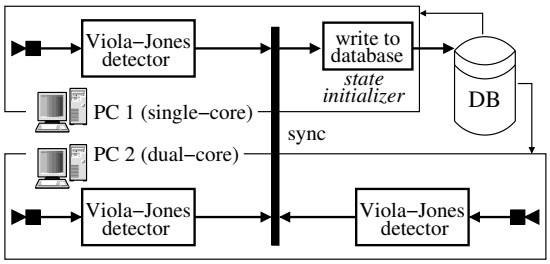

Figure 7: *Bootstrapper-only 3 camera system with a single- and a dual-core PC. The sensor handlers are omitted for clarity.*

Figure 7 depicts a system with 3 cameras distributed on 2 computers. Network communication and multithreading is automatically handled by the meta-system without user code. It is, however, the user's task to configure the system in a way that exploits processing power intelligently. In the example of Figure 7, PC2 is powerful enough to handle two detectors, therefore two detectors are delegated to it. PC1 handles only one detector and the state initialization.

If *state initializer* becomes computationally too complex, it is probable that PC1 will experience framedrops. In this case, it is possibly beneficial to move this plugin to a third computer. Please note that this requires only to redefine the network placement of the plugin in the 'BAT' configuration file, without the need to write a single line of code.

4.3. Model- and view-based 6DOF tracker for multiple gestures

Figure 8 depicts a tracking system that tracks distinct hand gestures in 6DOF. This system has three inputs, two color cameras and a near-infrared (NIR). Thus, it also serves as an example for sensor fusion. The system has both a bootstrapper and a tracker. The bootstrapper uses solely data from one color input and initializes the user handstate based on an appearance database. It also detects and updates the different gestures, which avoids the loss of tracking for the model based tracker in the case of gesture change.

The tracker is model based and optimizes the 6DOF handstate for every frame. For the color cameras the hand area is segmented on the GPU where also an XOR error function with the model fit is evaluated. The PC with the NIR input does not do feature extraction and evaluates the fit of a current hypothesis based on the edge distances between the model and the NIR image.

As the bulk of the tracking happens only in graphics hardware, the first PC has CPU time to search for best-fit initialization in the appearance database.

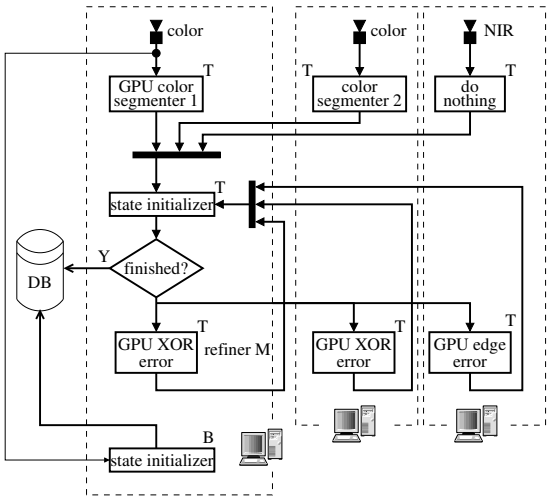

Figure 8: *6DOF multi-gesture tracker distributed on 3 PCs. Plugins with 'T' beside them belong to the tracker, the bootstrapper (B) consists only of a state initializer. For explanation cf. Section 4.3.*

5. Conclusions

We presented 'BAT', a framework that alleviates problems connected to implementing distributed (in the parallel execution sense) tracking systems. The motivation for such systems is that in some tracking tasks, like visual HCI, tracking framerate is of utmost importance. By parallelizing execution, one has the ability to exploit all the processor cycles available, not only in a multithreaded but also in a multicomputer sense.

Implementing such distributed systems involves a lot of infrastructure level code considering *e.g.* network communication or thread synchronization. 'BAT' takes care of these problems and allows researchers to concentrate on algorithm-level system development. The plugins implemented for 'BAT' are in fact reusable research results that can be easily combined in different ways. This encourages experimentation and accelerates system prototyping.

We plan to release 'BAT' under the FreeBSD software li-

© The Eurographics Association 2008.

cense. For more information, plase visit the project website: http://cg.cs.uni-bonn.de/project-pages/BAT/.

References

[AFM*06] ALLARD J., FRANCO J.-S., MENIER C., BOYER E., RAFFIN B.: The grimage platform: A mixed reality environment for interactions. In *ICVS '06: Proceedings of the Fourth IEEE International Conference on Computer Vision Systems* (Washington, DC, USA, 2006), IEEE Computer Society, p. 46.

[AGL*04] ALLARD J., GOURANTON V., LECOINTRE L., LIMET S., MELIN E., RAFFIN B., ROBERT S.: Flowvr: a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004* (Pisa, Italia, August 2004).

[AIM04] AHRENBERG L., IHRKE I., MAGNOR M.: A mobile system for multi-video recording. In *1st European Conference on Visual Media Production (CVMP)* (2004), IET, pp. 127–132.

[CES04] CULLER D., ESTRIN D., SRIVASTAVA M.: Guest editors' introduction: Overview of sensor networks. *Computer 37*, 8 (Aug. 2004), 41–49.

[dC06] DE CAMPOS T.: *3D Visual Tracking of Articulated Objects and Hands*. PhD thesis, Oxford University, January 2006.

[DSVG03] DOUBEK P., SVOBODA T., VAN GOOL L.: Monkeys — a software architecture for ViRoom — low-cost multicamera system. In *3rd International Conference on Computer Vision Systems* (April 2003), Crowley J. L., Piater J. H., Vincze M., Paletta L., (Eds.), no. 2626 in LNCS, Springer, pp. 386–395.

[EBN*07] EROL A., BEBIS G., NICOLESCU M., BOYLE R., TWOMBLY X.: Vision-based hand pose estimation: A review. 52–73.

[Fra04] FRANÇOIS A. R.: 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).

[Gav99] GAVRILA D. M.: The visual analysis of human movement: A survey. *Computer Vision and Image Understanding: CVIU 73*, 1 (1999), 82–98.

[LZT06] LITOS G., ZABULIS X., TRIANTAFYLLIDIS G.: Synchronous image acquisition based on network synchronization. *cvprw 0* (2006), 167.

[MFN06] METTA G., FITZPATRICK P., NATALE L.: Yarp: Yet another robot platform. *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics 3*, 1 (2006).

[MHK06] MOESLUND T. B., HILTON A., KRÜGER V.: A survey of advances in vision-based human motion capture and analysis. *Comput. Vis. Image Underst. 104*, 2 (2006), 90–126.

[RMTHS*01] RUSSELL M. TAYLOR I., HUDSON T. C., SEEGER A., WEBER H., JULIANO J., HELSER A. T.: Vrpn: a device-independent, network-transparent vr peripheral system. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2001), ACM, pp. 55–61.

[RS01] REITMAYR G., SCHMALSTIEG D.: Opentracker-an open software architecture for reconfigurable tracking based on xml. In *VR '01: Proceedings of the Virtual Reality 2001 Conference (VR'01)* (Washington, DC, USA, 2001), IEEE Computer Society, p. 285.

[TBAR05] TSOUMAKOS D., BITSAKOS K., ALOIMONOS Y., ROUSSOPOULOS N.: A framework for distributed human tracking. In *PDPTA* (2005), pp. 863–868.

[TLMS03] THEOBALT C., LI M., MAGNOR M., SEIDEL H.-P.: A flexible and versatile studio for synchronized multi-view video recording. *Proc. IMA Vision, Video, and Graphics 2003 (VVG'03), Bath, UK* (July 2003).

[VJ01] VIOLA P., JONES M.: Robust real time object detection. In *SCTV01* (2001).

© The Eurographics Association 2008.