Max–Planck–Institut für biologische Kybernetik
Max Planck Institute for Biological Cybernetics

Technical Report No. 123

The MPI VideoLab - A system for high quality synchronous recording of video and audio from multiple viewpoints

Mario Kleiner[1], Christian Wallraven[1] &
Heinrich H. Bülthoff[1]

May 2004

[1]Department Bülthoff, Max Planck Institute for Biological Cybernetics, Spemannstr. 38, 72076 Tübingen, Germany.
E-mail: mario.kleiner@tuebingen.mpg.de

This report is available in PDF–format via anonymous ftp at ftp://ftp.kyb.tuebingen.mpg.de/pub/mpi-memos/pdf/TR-123.pdf. The complete series of Technical Reports is documented at: http://www.kyb.tuebingen.mpg.de/bu/techr/

The MPI VideoLab - A system for high quality synchronous recording of video and audio from multiple viewpoints

Mario Kleiner, Christian Wallraven & Heinrich H. Bülthoff

Abstract. The MPI VideoLab is a custom built, flexible digital video- and audio recording studio that enables high quality, time synchronized recordings of human actions from multiple viewpoints. This technical report describes the requirements to the system in the context of our applications, its hardware- and software equipment and the special features of the recording setup. Important aspects of the hardware and software implementation are discussed in detail.

Keywords: multi-viewpoint video recording, distributed system

---

1 Introduction

This report describes the MPI VideoLab, which is a flexible system for high quality video and audio recordings of human actions from multiple viewpoints. There are a number of research areas where such a setup can be applied:

- **Psychophysics:** Studies on recognition of facial expressions and human actions. By providing annotated databases of facial expressions and human actions, this setup can help to address fundamental questions of perceptual research in a controlled experimental setting.
- **Psychophysics:** Studies on viewpoint generalization in recognition tasks: How sensitive are humans to *changes* in viewpoint when having to recognize spatio-temporal patterns of human actions (facial expressions, gestures, body language, etc.)?
- **Psycholinguistics:** Studies on multi-modal context effects of facial expressions: How do sound and vision interact during successful communication?
- **Computer Vision:** Controlled capture of human actions from multiple viewpoints. This can be used, for example, for dynamic, three-dimensional tracking and reconstruction of faces and gestures. The use of multiple cameras allows us to perform these tasks more efficiently, as multiple viewpoints introduce important constraints for interpreting the two-dimensional projections of three-dimensional points.
- **Computer Vision:** Recognition of human actions and facial expressions. By recording and annotating several controlled databases of human actions and facial expressions, the setup allows us to create training and testing sets for evaluating recognition algorithms from computer vision and machine learning.
- **Computer Graphics:** Baseline for computer animation. Traditional computer animation, for example of three-dimensional facial expressions (avatars), relies crucially on a "ground truth". This setup can provide not only a static baseline from multiple viewpoints, but also a dynamic baseline of how the facial features move between two points in time.

In this report, the focus is on describing the *technical details* - hardware as well as software - of the VideoLab, rather than going into more detail about possible applications. From the above list, however, it becomes clear that such a setup has to fulfill a number of important requirements in order to be a useful research tool.

1.1 General requirements

More specifically, from the list of applications, we can identify six critical technical requirements for the system.

The first and most fundamental requirement is that all cameras capture video frames synchronously in time, with synchronization maintained over extended recording periods. The capture times of corresponding frames from different cameras must not deviate from

each other by more than a few microseconds in order to be suitable for multi-view computer vision algorithms and precise psychophysical investigations. In addition, recorded sound should stay synchronized to the video stream with millisecond precision.

Second, to accomplish color based tracking and the creation of high-quality visual stimuli, the video images need to be captured in color at a high image resolution under well controlled lighting conditions. The footage should be free of interlacing, motion blur, video compression artifacts or image noise.

Third, to capture and reconstruct the dynamics of human facial expressions or articulated movements without motion blur, we need to have exposure times below 5 milliseconds and sometimes frame rates higher than the typical 25 full frames per second, that are often used for conventional video recordings.

Fourth, moving video stimuli for experiments are usually presented on standard computer equipment, either as sequences of single images that are presented with controlled timing, or as standard video files in MPEG, QuickTime or AVI format. Computer vision and image processing algorithms need digital input as well. Therefore, in order to make handling and archiving of the video footage as easy and straightforward as possible, all video footage should be stored in a digital file format on regular computer filesystems instead of analog video tape. Fully digital storage also prevents quality loss due to conversion from analog to digital footage and reduces the need for labor intensive and error prone human intervention in the conversion- and postprocessing process.

The fifth requirement is that the handling of the system should be easy to learn and the setup should be easy to use, reliable and robust against operator errors or technical error conditions. This is due to the fact that the recording setup will be often used by people without a technical background in video processing, and typical recording sessions often consist of up to one hundred single recordings.

Finally, a desirable property is easy and relatively cheap extensibility of the setup to a higher number of cameras or audio channels.

1.2 Limitations of common video equipment

Cameras and recorders for formats like VHS, S-VHS, DV, DVCPro, Betacam or Digital Betacam, do not provide all the technical features that are needed to meet the requirements mentioned above. Most video cameras can not control the start of frame capture with microsecond accuracy via an external trigger signal, making it very difficult to simultaneously control and synchronize multiple cameras. They usually operate at a fixed frame rate of e.g., 50 fields per second (European PAL standard) or 60 fields per second (NTSC standard). Control of exposure time, focus or color gain is usually handled by automatic black box circuitry, making it difficult to control for these parameters. Video frames are usually not captured as full images, but instead as time-sequential pairs of even and odd fields: One field only contains the even numbered scan lines, the successive field contains the odd numbered scan lines. This leads to unwanted interlacing artifacts, especially when fast horizontal movements occur in the scene.

Another problem of standard recording equipment is loss of image quality. Analog recording systems suffer image degradation due to “pollution” of the video signal during transmission over long cables, quality loss caused by aging of magnetic storage tapes and loss caused by the analog to digital conversion process during readout. Digital transmission and storage systems usually have a much higher signal to noise ratio and better means of error correction by design, but they have to apply lossy image compression algorithms to the video data to reduce storage space and bandwidth requirements, thereby artificially degrading image quality. While the selected trade off between image quality and storage space is well suited for consumer video applications and broadcast transmission over television channels with their limited bandwidth, it is not acceptable for our range of applications.

2 System implementation

To overcome the restrictions of standard video equipment, we combined specialized video recording hardware that is targeted to industrial machine vision application with off-the-shelf computer hardware and custom video recording software to build a high performance distributed video recording system.

2.1 Recording hardware

Our system is designed as a distributed computer cluster of video and audio recording nodes (see figure 1). Each recording node consists of a specialized digital video camera, a specialized frame grabber and - optionally - a sound card, which are attached to a standard Intel-x86 compatible PC with fast hard disks. The computers of the nodes are connected with each other and with a control computer as well as a file server via a standard 100 MBit Ethernet local area network. The frame grabbers of all nodes are connected to each other for the transmission of an electronic trigger signal that allows for high precision synchronisation of frame capture between the nodes. Distributed control software (see section 2.2) presents the cluster as a single unified system to the user and application programmer.

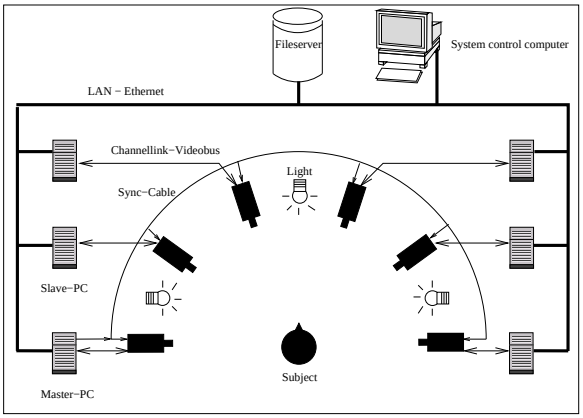

Figure 1: Overview of the video setup, showing a typical arrangement of the cameras and lights and the connections between different recording nodes.

Currently (as of April 2004), we use six recording nodes, each equipped with one camera. One of the six nodes additionally contains a stereo sound card, giving us the ability to record audio signals on two channels with compact disc audio quality. The system can be extended to any number of cameras and audio channels by attaching additional recording nodes.

2.1.1 The cameras

Our current system uses A302bc cameras from the German company Basler [1]. This camera model has the following properties:

- A single chip, progressive scan 1/2" CCD sensor, using a Bayer color filter mask [4] for color interpolation.
- Maximum image resolution 782 x 582 pixels. Square pixels of size 8.3 μm by 8.3 μm.
- Programmable 8 or 10 Bit analog to digital conversion in the camera with programmable gain and offset.
- Programmable region of interest (ROI).
- Programmable exposure time between 10 microseconds and 1000 milliseconds.
- Programmable frame rate of up to 60 full frames per second.
- Digital output of CCD sensor intensity measurements over a Channel Link video bus.
- Exposure interval and readout of the CCD chip can be triggered by attached frame grabber.
- C-Mount connector for standard optical lenses.
- Integrated Infrared - Cut filter to mask out incident infrared light.

The progressive scan CCD sensor reads out and transmits images as full frames, thereby preventing any kind of possible interlacing or tearing artifacts. As the conversion of CCD sensor measurements from analog to digital values is done by an internal converter in the camera immediately after readout from the CCD chip, transmission of video data from the cameras to the frame grabbers already happens in digital form, making the captured video data robust against degradation by electronic noise in the cables. All relevant acquisition parameters of the cameras are programmable either via the Channel Link video bus from the frame grabber or via a serial RS-232 link from the responsible computer. The high frame rate together with the ability to select very small exposure times and high signal amplifier gains allows us to capture fast movements without significant motion blur.

Another option for an image sensor would have been CMOS (complementary metal oxide semiconductor) technology instead of CCD (charge coupled device) technology. These imaging sensors are cheaper, allow for smaller cameras, as the analog to digital converters are already integrated into the sensor, and require less electric power (See [2] for an in-depth explanation and comparison of the two technologies). Another interesting feature is the ability to have random access to sensor pixels or regions of the chip: If one is only interested in small portions of an image, one can just read out that areas instead of having to read out a full frame like with CCD chips. This allows for higher frame rates. Their big disadvantage with respect to CCD sensors is a smaller sensitivity to light and a higher noise level.

As we wanted to have the highest possible image quality and a high light sensitivity for recordings with short exposure times, CMOS sensors were ruled out in favor of CCD sensors.

2.1.2 The frame grabbers

The frame grabbers are "microEnable MXA36" from the German company Silicon Software [3]. Each frame grabber is attached to its recording computer via a standard 32 Bit, 33 Mhz PCI bus and to the cameras via a Channel Link video bus cable of 5 meters length. The frame grabber has 512 KB of SDRAM memory and a "Xilinx XC4036XLA" Field programmable gate array (FPGA). The FPGA circuitry on the frame grabber is freely programmable in the "Very high speed integrated circuit Hardware Description Language" (VHDL). The implementation and behavior of the frame grabber hardware is therefore not hard-wired as on conventional frame grabbers, but only deter-

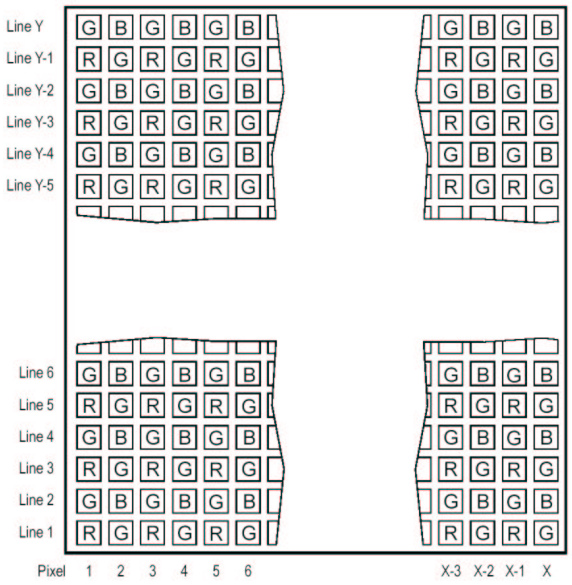

Figure 2: The Bayer color filter array mounted on our cameras CCD chip. R=Red filter, G=Green filter, B=Blue filter. (Picture from the A302bc camera users manual.)

mined by the VHDL firmware (called "Hardware applet") that is downloaded to the frame grabbers FPGA during system initialization by our recording software. With an appropriate hardware applet, the frame grabber can act as a realtime image preprocessor for low level image processing operations like e.g., image convolution, low-pass filtering, edge detection and color conversions.

Lossless data compression:

The vendor-supplied default hardware applet would use the FPGA to perform on-the-fly computation of RGB pixel color values by interpolation from the raw pixel intensity measurements, a technique known as demosaicking (see [5] for a comparison of different demosaicking techniques). It would also perform color correction before transmitting the video data to the computers for writeout to the hard disks. Similar functionality is hard-wired into standard video cameras and conventional frame grabbers.

We use the programmability of the frame grabbers to *prevent* any kind of image preprocessing of the measured CCD pixel intensity data by the camera or frame grabber hardware. That is, our hardware applet ensures that the original CCD pixel intensity data recorded by the cameras is left unaltered by the frame grabber, so our recording software is able to store this raw data to the hard disks of the attached recording computer. Storing raw data has two important advantages for our application:

The most important advantage is a reduction in the amount of required disk storage space as well as required hard disk write bandwidth by avoiding the demosaicking step: Single chip CCD cameras only have one sensor element per captured image pixel. At each image pixel location, they can only measure either an intensity value in the "red" part of the spectrum, or in the "green" part of the spectrum, or in the "blue" part of the spectrum, depending on the color filter in front of the sensor element (see Figure 2 for the distribution of color filter elements over the CCD matrix of our cameras). As each output pixel in the final captured image needs to have a red-, green- and blue-component, the two missing color values for a specific pixel location are calculated as a weighted sum over the appropriate color component intensity values that are measured at neighboring sensor locations. Let us illustrate this interpolation, e.g., for the output pixel that is computed for the 3rd column in the 2nd line of the CCD sensor array: As can be seen in figure 2, the sensor element at position $(x, y) = (3, 2)$ is equipped with a green color filter, therefore providing a direct measurement (denoted as $I(x, y)$) for the value of the green component $G(x, y)$ of the output color, so $G(x, y) = I(x, y)$. The value of the blue component is interpolated by taking the average over the measurements of the neighboring sensor elements at positions $(x-1, y)$ and $(x+1, y)$, because these sensors are equipped with blue color filters, so $B(x, y) = \frac{I(x-1,y)+I(x+1,y)}{2}$. The red component is computed as average over the sensor readings at positions $(x, y-1)$ and $(x, y+1)$, where red color filters are installed in front of the sensors, so $R(x, y) = \frac{I(x,y-1)+I(x,y+1)}{2}$. Depending on the position (x, y) of the output pixel (x position even or odd, y position even or odd) one encounters four different configurations of color filters in the three by three neighborship, therefore four different sets of equations for color interpolation are required. The equations given for our example position $(x, y) = (3, 2)$ apply for all pixels with odd x position and even y position. Equations for the other cases can be easily derived from figure 2 and are left as an exercise to the reader.

The demosaicking step outputs three color values $R(x, y), G(x, y), B(x, y)$ for each physically measured value of pixel intensity $I(x, y)$, effectively triplicating the amount of image data that needs to be processed and stored to disk. If we would write out the demosaicked digital video data stream at our highest image resolution and frame rate to the hard disks, we would have to write out a data volume of 782 * 582 pixels * 3 color components per pixel (Interpolated RGB color values) * 1 Byte per component * 60 frames per second = 78.2 Megabytes per second,

which is not feasible, because conventional hard disks are not fast enough to write out a data stream at 78 MB/s. High-end RAID arrays could handle such a writeout bandwidth, but they would be very expensive to buy and maintain, especially if one needs a RAID unit for each video recording node.

Storing the unfiltered data to the hard disks reduces the required disk write bandwidth to 26.05 MB/s plus additional 0.1 MB/s for each recorded audio channel if sound recording is enabled. Our specially configured fast IDE hard disks (see Section 2.3.2) are able to sustain an average write data rate of 39 MB/s for multiple hours, allowing for very long uninterrupted recordings. As a side effect of storing the unfiltered data, we also save a factor of three of storage space.

The second advantage of using raw sensor data as our native video data storage format is the possibility to reapply different postprocessing algorithms e.g., color calibration, as an offline postprocessing step any time after acquisition of the video data. See sections 2.4.2 and 2.4.3 for examples of postprocessing algorithms.

Accurate camera synchronization:

The start of frame capture of each frame grabber can be triggered externally by a digital synchronisation signal and each frame grabber can act as a signal generator for the creation of such sync signals. We use this capability to ensure accurate synchronisation of frame capture between all recording nodes: One of the frame grabbers (the "master") is programmed by our recording software to act as a sync signal generator. Whenever the master frame grabber starts capture of a new video frame, it generates and distributes a sync signal via a dedicated cable to all other frame grabbers. These "slave" frame grabbers start capture of their next video frames upon arrival of the external sync signal. This way, capture of corresponding video frames happens simultaneously on all recording nodes with a maximum jitter of frame capture timing between the different nodes of less than 1 microsecond. This accuracy is sufficient for even the most demanding computer vision and graphics applications.

2.1.3 Computer system

The underlying computer system of the video setup consists of off-the-shelf hardware, driven by a customized version of the free GNU/Linux operating system.

Operating system:

We have chosen to run the computers with different versions of the free GNU/Linux operating system [6], because Linux has several advantages as a realtime cluster operating system.

- Flexibility and transparency: It was easy to understand and customize all relevant aspects of the working of the Linux operating system even at a low level, because all components are either available as source code (e.g., operating system kernel, device drivers) or implemented as easily modifiable shell scripts and human readable text configuration files (e.g., bootup-, control- and shutdown scripts). Easy access to the inner workings of the system, together with very detailed technical documentation that is available on the Internet, allowed us to modify the system to fit our special needs and was especially helpful for debugging and fine-tuning the application.

- Robust realtime behavior: Our application makes challenging demands onto the operating system regarding consistent disk write throughput and consistent maintenance of tight application timing deadlines (see Section 2.3.1 for details). Missing this deadlines even by a few milliseconds would lead to loss of time synchronization between the different recorded video- and audio streams or to dropped video frames. Our customized Linux system, albeit using a standard Linux 2.2.16 kernel without special realtime extensions, showed very reliable timing, with no dropouts at all, during uninterrupted test recordings of up to 3 hours. Simple benchmarks under Microsoft Windows NT indicated that it would have been difficult at best, to reach the same level of realtime reliability.
  If standard Linux would have been unable to meet the realtime demands of our application, we could have used a hard realtime version of Linux, e.g., "Realtime Linux" from FSMLabs [7], or Microsoft Windows CE 3.0, a realtime version of Microsoft Windows. Both systems, according to their distributors, provide fully deterministic thread scheduling and worst case latencies for thread scheduling and interrupt handling of less than 50μs on comparable hardware. A problem with such systems would have been lack of device driver support for the frame grabbers, requiring costly porting of the device drivers by our vendor[1], and a more complex design of our recording application due to the more involved programming models of these realtime operating systems.

- Stability and robustness: Stable operation of the system for multiple hours under very high load

[1] As of May 2004, our frame grabber vendors provide device drivers for Realtime Linux, but in 2001, when we started development of the system, this was not the case.

is crucial for typical recording sessions with our setup. Debugging a distributed parallel system, like our application, is challenging as well. Therefore it is important to have an operating system that works reliably under high loads and that is very robust against operator errors and application malfunctions. Linux has a very good reputation in this area and fulfilled our expectations.

- Ease of maintenance: All important management tasks, even system installation on new hardware, can be easily automated via scripts. One can perform all system administration tasks from a remote console over the network, without the need to attach keyboards or displays to the single machines.

- No licensing costs: Linux can be downloaded, installed on as many computers as required free of charge, and customized in accordance with the GPL license [11].

The exact type of Linux systems and necessary modifications to the system are described in more detail below, together with the description of the computer hardware.

Computer hardware:

The computer hardware currently consists of six identical recording nodes, one system control computer for the user and a dedicated fileserver for persistent storage of the recorded data and data postprocessing.

- Our recording computers are standard Intel PCs with Intel Pentium-III processors, running at 800 Mhz clock speed with 256 MB of RAM. The computers use 100 MBit ethernet adapters to connect to the control computer and fileserver. Each computer is connected to its associated frame grabber and - optionally - a standard sound card[2] via the built-in PCI bus, and to its associated camera via a standard RS-232 serial link. The machines are "headless", they are completely controlled, including powerup and shutdown, via network without any need for a keyboard, mouse or display. Each computer contains a fast IBM DTLA-307075 75 GB IDE hard disk drive and a IBM Deskstar IC35L120AVVA07-0 115 GB hard disk drive[3], which was added later to improve write performance and to increase storage capacity by combining both drives into a virtual RAID-0 drive (see Section 2.3). The local storage capacity for video data on each node is currently 180 GB, allowing for uninterrupted video recordings of up to 2 hours at maximum image resolution and frame rate.
  The operating system is a customized version of RedHat-Linux 7.0, running Linux kernel version 2.2.16. The system has been modified to simplify maintenance of the recording cluster and to guarantee the good realtime performance and high disk write throughput crucial for our application. See Section 2.3 for a detailed explanation of performance enhancements.

- The system control computer is a standard Intel Pentium-4 system running at 2.2 Ghz clock speed. It is equipped with 1 GB of system RAM and a GeForce 4 graphics card. It is used to display the graphical user interface for control of the recording setup by the user. The operating system is a standard Linux distribution, currently RedHat-Linux 7.2, running Linux kernel 2.4.20.

- The fileserver is a Dell PowerEdge 2550 server, equipped with two Pentium-III Xeon processors, each running at 1.1 Ghz clock speed. It has 1 GB of onboard system memory. The operating system is RedHat-Linux 7.2, running Linux kernel 2.4.20. The fileserver is used for reliable persistent backup of recorded video data from the video recording nodes and for providing the data to in-house client computers. Currently it uses a hardware RAID level 5 high performance SCSI - disk array from Dell with a disk capacity of 600 GB for data storage. The current RAID-5 configuration protects data integrity in case of hardware failure of one drive. Redundant internal power supplies, a connection to an uninterruptible power supply and the Linux Ext3 journaling filesystem provide protection against possible data loss or data corruption in case of power outage, unclean system shutdowns or hardware failures. The RAID array is made available via the Linux Logical Volume Manager (LVM) to allow for easy extensibility of the storage capacity, if needed. The fileserver runs multiple server processes to export the video data to client computers:

[2] Currently, we use a single Soundblaster compatible "Ensoniq 5880 AudioPCI" sound card with two channels, but the system would also support high-end multichannel PCI sound cards, if needed.

[3] The exact model names are provided simply as a reference for a drive combination that is known to work reliably. Any other combination of modern IDE drives with the same or better specifications regarding minimum, maximum and average data throughput and minimum capacity should work as well.

– A NFS V2 and V3 server daemon for serving in-house, Unix compatible client systems like Linux, FreeBSD, SGI-Irix and Apple MacOS-X.
– Version 2 of the Samba SMB server [15] for data access from Microsoft Windows client systems.

2.2 Software framework

The software of the VideoLab currently consists of multiple applications which will be described in more detail in the following sections:

- The distributed recording software "vlgui" for multi-view video- and audio recording, using the video setup. This is an interactive application with a graphical user interface for online control of the distributed setup during preparation and running of the recording sessions.
- A specialized movie player "vlmovieplayer" for playback of the recorded video streams, giving access to the special properties of the multi-view video- and audio streams. It is completely controlled with a GUI and also allows for online selection and transcoding of the video streams into standard movie file formats like e.g., MPEG and AVI for export into common video editing, playback and postprocessing applications.
- A postprocessing tool "vlvideofile2ppm" for different kinds of video- and audio postprocessing operations. This is a command line tool for batch processing of large amounts of data, controlled by shell scripts.
- Specialized applications for camera calibration, face tracking, facial texture extraction and manipulation from recorded video. These applications are mostly used for the generation of special stimuli for psychophysical investigation of human face perception. A discussion of this applications is beyond the scope of this report, see [20] for some details.

All applications are written either in the C or C++ programming language for the GNU/Linux operating system on Intel x86 compatible hardware. Applications that use the VideoLab, or data recorded with the VideoLab, do so by linking against the "libvideolabbase" library, our custom system library of C++ classes for control of the video setup and access to the video data.

The following sections will give an overview of the "libvideolabbase" library, a detailed explanation of the working of the parallel distributed recording process and an overview of the different postprocessing applications.

2.2.1 The libvideolabbase - library

All functionality for control of the distributed video recording system and for access to - and manipulation of - recorded video- and audio data is implemented as a collection of C++ classes in the libvideolabbase library. The library is a shared library, VideoLab applications are dynamically linked against the latest version of the library at application startup, therefore the implementation of the library can be improved (e.g., performance enhancements or bug fixes) without the need to recompile the different applications that use the library.

Encapsulating system control functions and data access functions into a library of C++ classes has three advantages:

- It hides complexity from the application programmer: Application programmers do not need to know much about the data format of the video- and audio files on the filesystem or about the technical implementation of the recording setup. They are presented with a clean, consistent and easy to learn C++ programming interface for performing low level operations like e.g., starting the recording hardware, selecting and setting up cameras and audio channels for a recording, setting up exposure time or frame rate, starting or stopping of a single recording and creation, reading and writing of video files.
- Functionality that is potentially useful or necessary for multiple applications can be shared by implementing it in the library.
- System developers are able to make changes "under the hood" of the system, e.g. bug fixes and performance enhancements, without affecting the behavior of previously written applications.

The library currently contains the following classes:

- *VLE*: A class for the static definition of status- and error codes. The VideoLab library has extensive self diagnostic routines and error handling functions to guard against operator errors and hardware problems. In case of error, it writes detailed error descriptions into system log files to aid system developers in debugging. It also provides human readable error messages, as well as step by step troubleshooting tips, to the calling application. These messages are preformatted and intended for presentation to the user, e.g. via output into dialog boxes. The library also returns symbolic error- and status codes for high level error handling routines in the calling applications. These codes are defined in the VLE class.

- *VLProject*: The VLProject class is a "container" for all settings that apply globally to all views of a single multi-view recording. It provides methods for querying and changing such global settings like e.g., frame rate, number and selection of cameras for the respective recording and total frame count for the current recording. It contains methods for creating, cloning, deleting and accessing recordings as a whole, for simultaneous seek operations in all video files of a video recording and - most importantly - for accessing the single video files of the different cameras of a multi-view recording.

- *VLVideofile*: The VLVideofile class represents and implements all properties and procedures that are not global to a multi-view recording, but local to a single video file created by a specific camera. That is, the settings that are not shared between all cameras and video streams of a multi-view recording, but that are usually different for different cameras of the recording. Examples of such per-camera or per-videostream settings and methods are e.g., exposure time, camera amplifier gains, color calibration parameters, region of interest or the sound settings of associated sound channels like volume, sampling rate and number of sound channels.
  The most important method for all data postprocessing applications is the *getFrame()* method: It reads unfiltered raw data for a single video frame from a video datafile on the hard disk drive, applies different post processing operations to it and returns the video frame as a standard 24 Bit per pixel, RGB true color bitmap image for display or further processing by the calling application. As our system stores video data in a space-efficient raw data format (see Section 2.1.2), the first step is conversion of the raw data into the standard RGB pixels format, where each image pixel is represented as a sequence of three one-byte numbers, each number encoding the intensity of one of the color channels. This is done by a software implementation of a demosaicking color interpolation algorithm in the VLVideofile class (see Section 2.1.2 for a basic explanation of the algorithm). The same routine performs on-the-fly color correction of images, checks for saturated pixels due to over- or underexposure of single camera sensor elements and applies a "deflickering" operation to video data, if required (see Section 2.4.2). Another feature is on-the-fly image rotation by 90 degrees to allow for recordings in portrait format by tilting the camera by 90 degrees around its optical axis. The routine is implemented as a mixture of hand optimized C++ and Pentium-MMX assembler code[4], as it is crucial for the performance of all postprocessing applications.

- *VLRecorder*: The VLRecorder class is the application programmers interface to the recording hardware. It provides all necessary methods for querying the state of the recording hardware, setting up and controlling the recording hardware during a recording session. It also implements the single central control instance that communicates with all recording nodes of the cluster, thereby enabling the cluster to present itself as a single unified system to the application programmer.

All classes except the *VLRecorder* class are written in a portable way, so postprocessing applications that use the library can be easily ported to other operating systems like Unix, Windows or MacOS-X. Only the core recording functionality implemented in the *VLRecorder* class is dependent on special features of GNU/Linux and therefore not easily portable to other operating systems.

2.2.2 Filesystem structure of projects

The VideoLab stores recorded video data, audio data and meta information of each recording (referred to as a single "*project*") in a hierarchy of directories and files, that changes between the recording stage of a project and the offline processing stage of a project, because these stages have different requirements on the storage place and format:

- During the recording stage, highest possible disk write performance has top priority: Each camera produces up to approximately 27 MB/s of data per second. To allow such a high writeout rate for extended amounts of time, the video data needs to be stored on fast local hard disks that are installed in each single recording computer. The hard disks and filesystems need to be specifically tuned to allow for consistently high throughput (See section 2.3). This tuning leads to reduced protection against data loss in case of system malfunctions. Distributing data across multiple computer nodes also makes data backup and access by secondary applications more difficult.

- The offline processing stage has three important requirements:

  – Easy data access: Data should be structured in a format that allows easy access by

[4] When compiling this class for target operating systems other than Linux, the MMX assembler code gets replaced by slower, but portable C++ code.

secondary applications (e.g., video players, video converters), to simplify the design and implementation of this applications. Storing all data in a central place facilitates simple data access.

– Data protection: As recording sessions are potentially expensive and time consuming, special care has to be taken to protect the recorded data against potential loss or corruption e.g., due to system malfunctions or user errors.

– Flexibility: VideoLab recordings produce a huge amount of data, e.g., over 9 GB per minute, when recording with six cameras at full frame rate and resolution. Working with such large amounts of data poses challenging problems in managing the data. It should be easy to move the data around between different storage places for backup and archiving purposes.

Our system implements two different file layouts for the recording- and postprocessing stage with an automatic conversion procedure from the recording layout to the postprocessing layout.

File layout during recording stage:

When a new recording project is created, the system creates a distributed hierarchy of directories and files.

- On the fileserver, a project directory with the name of the recording project is created in a location selectable by the user. Metafiles are created in the project directory, which store information global to the recording and not specific to a single camera, e.g. all the information contained in the *VLProject* class is stored in a file with extension *.vlprojectfile*.

- On the local hard disks of each recording computer, a subdirectory is created, whose name consists of the project name and an unique identifier for the camera to which the computer is attached. The subdirectory contains one header file that stores all settings that are specific to a single cameras recording, such as exposure time, region of interest - all the information that is represented by the *VLVideofile* class in the running application. During recording, the recorded video- and audio data is written to files in that subdirectory: The recorded data is split up into numbered files of a maximum size of 2GB.

The metadata files on the fileserver therefore contain a high level description of the recording, as well as the information to localize the actual recorded data on the single recording computers, while the local recording machines contain the video data on their local hard disks for fast write access.

File layout after recording stage:

At the end of a recording session, a backup script is started, which moves the video data directories from the local recording computers into subdirectories of the project directory on the fileserver and adapts some meta information to take the new data layout into account.

After the backup operation, all data of a project is stored in a central location - on the fileserver - in a directory hierarchy that allows easy access to - and backup of - the recorded data: The project as a whole can be easily accessed by secondary applications, utilizing the *VLProject* class. The whole project directory could be moved or backed up to other computers and media. If one is not interested in the whole multi view recording, but only in video data of a single camera, one can easily access a single cameras data with the help of the *VLVideofile* class. Single view data can be moved or backed up to other locations just by moving the data subdirectory of a camera that is contained in the project directory.

2.2.3 Implementation of the distributed recording software

The following paragraphs provide some details on the implementation of the distributed recording software.

System control architecture

The VideoLab recording software is organized as a distributed parallel system. It consists of a single master process that runs on the VideoLab control computer and interacts with the user, and multiple slave processes, one slave process running on each computer node in the recording cluster.

The master process can be any C or C++ application that is linked against the libvideolabbase - library. This application has to implement some form of high level control logic for running a recording session, possibly guided by interaction with the system operator. Actions of the recording setup, for example, change of active camera set, modification of recording parameters, creation and management of video files, or start and termination of video- and audio recordings, are requested by calling the appropriate methods of the VLRecorder class with appropriate parameters. These methods check the requested operation and its parameters for validity, taking the current state of the recording setup into account, and issue corresponding *remote procedure calls (RPC)* over the network to the slave processes of all relevant recording computers. Once

started, each slave process passively waits for RPC requests from the master in an loop until application shutdown. After processing a request, a status code is returned to the master, confirming successful completion of the request, possibly returning some additional status information about the recording node. If a request fails on a recording node due to some local error condition, the node's slave process performs error handling to get the node back into a well defined and safe state if possible, and returns an error code to the master process to allow it to perform high level error handling, if necessary.

Requests for operations that affect multiple recording nodes are sent out and handled by the VLRecorder class in a parallel, non-blocking fashion. This asynchronous RPC mechanism - although requiring more complex error handling - makes it possible to parallelize expensive operations between all recording nodes involved. The total time needed for performing an operation is therefore kept independent of the number of recording nodes involved, enabling the system to scale up to a potentially unlimited number of cameras.

The basic infrastructure for communication between the master process and its slave processes and for management of the distributed software system (startup and termination of slave processes on the recording cluster nodes, cluster management, application health monitoring[5]) is provided by the *CPPVM-Library* [8]. This C++ library is available as free software under the *LGPL-License* [10]. It implements a C++ interface to the popular *"Parallel Virtual Machine" PVM-Library* [9] for parallel distributed computing, which is also available as free software under the LGPL.

Implementation of a single recording node

As mentioned above, each recording computer is controlled by one slave process that is initiated by the *CPPVM-Library* at startup of the controlling master process. This slave process, called *pvmrecorder,* implements the realtime video- and audio recording process by controlling the hardware of the system and disk writeout of the recorded data.

Pvmrecorder is internally split up into three parallel threads of execution - one control thread for communication with the remote master process and handling of basic RPC operations, and two additional threads that implement the realtime recording loop.

The *control* thread is started at startup of *pvmrecorder*. After basic initialization and establishment of the network connection to the remote master process, it enters a command receive loop that is only left at termination of the *pvmrecorder*: It waits - blocked - for receipt of RPC requests from the master process. If a RPC is received, it acknowledges receipt of the RPC to the master process, executes the corresponding routines on the local recording computer, returns some status information back to the master process and reenters its command receive loop.
When asked by the master to start recording, it sets up the recording hardware, creates and sets up the two parallel recording threads and triggers start of their recording operation. Termination of a recording is performed by signaling the recording threads to stop their operation, waiting for them to finish, terminating the recording threads, resetting the recording hardware and performing related cleanup work.
As the actual capture and data writeout operations are done by the parallel recording threads, the *control* thread stays responsive to RPC requests from the master process while a recording is in progress, e.g. for status queries and stop requests.

The continuous recording operation is split up into two independent threads, a *grabber* thread and a *writer* thread:

- The *grabber* thread is responsible for direct communication with the data capture hardware - the frame grabber and the sound card. It communicates with the sound hardware via the *Open Sound System (OSS)* audio device drivers that are part of the standard Linux kernel and that allow fine-grained control of most available sound cards. Communication with the frame grabber hardware is done via a proprietary, binary only, device driver that is supplied by the frame grabber vendor. The implementation of the thread is basically an endless loop that consists of the following steps:

  1. Wait for arrival of video frames in the small circular memory buffer of the frame grabber. The thread sleeps for half the expected duration of a frame, then gets woken up by a timer interrupt of the system realtime clock. It polls the frame grabber driver for new video frames in its circular memory buffer and for error conditions. If neither of these events happen, it sleeps for another half frame duration and then repeats step 1.
     The frame grabber driver allocates a small

[5] Our system performs permanent periodic health checks, using CPPVM's diagnostic functions, to make sure that all nodes, processes and the network connections work properly. If one of the processes or computers would crash or the communication would fail, all non-affected processes would detect this condition and shut themselves down in a graceful way to prevent data loss or uncontrolled operation of the system.

memory buffer with a maximum capacity of eight video frames at startup. After start of video capture, the frame grabber hardware automatically captures video frames on arrival of external sync signals and transfers their data into this ring buffer, using *"Direct Memory Access" (DMA)* transfers without involvement of the systems CPU. The grabber thread has to read out and process new frames from this small buffer before the buffer overflows. While the grabber thread has to be fast enough to handle new frames in a time span of less than 16 milliseconds on average, the ring buffer can compensate for single, isolated interruptions of grabber thread operation of up to 128 milliseconds[6]. Such short interruptions can happen e.g., if the operating system has to perform internal book keeping work or if the *control* thread has to process RPC requests that arrive from the master process over the network.

2. When the first frame of a recording arrives, the sound recording hardware is started. As we use standard sound hardware, no hardware based synchronisation can be done. Instead we start sound recording after capture of the first video frame. We measured a lag between arrival of the first video frame and start of sound recording below 1.2 milliseconds, which is sufficient for our purpose. After start of sampling, the sound hardware uses DMA operations to store sampled sound data into a ring buffer, allocated by the sound driver.
3. Read out frame data from the frame grabber's ring buffer. Read out corresponding audio data from the DMA ring buffer of the sound driver. Reassemble it into one single data block.
4. Copy the data block into the next free slot of an in-memory ring buffer with a capacity of 100 MB (approximately 230 frames).
5. Notify the *writer* thread that new data has arrived in the big ring buffer.
6. Check for error conditions and abort with proper error handling, if so.
7. Repeat with step 1 for processing of next frame.

[6]These numbers apply to the maximum recording frame rate of 60 fps. At 60 fps, a single frame lasts 16 ms, 8 frames last 128 ms. Under normal operating conditions, processing of a single frame by the grabber thread takes less than 8 ms.

To summarize, the main task of the grabber thread is reading out video- and audio data from the separate, hardware driven, small DMA ring buffers, merging the video- and sound data blocks into single data blocks and transferring them into the larger memory ring buffer. Due to the limited capacity of the hardware driven DMA ring buffers, this process has to be very fast to prevent the small buffers from overflowing.

- The *writer* thread performs writeout of the data from the big 100 MB memory ring buffer into data files onto the local hard disks of the recording node. Video data files are split up into chunks of a maximum size of 2 GB. Whenever a data file has filled up with 2 GB of data, it is closed and an additional file is created for writeout of the next up to 2 GB data chunk. This splitting is necessary, because many common operating systems, filesystems, network protocols and applications do not support access to data files of a size bigger than 2 GB[7]. The *writer* thread gets notified by the *grabber* thread whenever new content is available in the memory ring buffer. It then issues write requests to the operating system to trigger writeout of the new data from the ring buffer. The 100 MB ring buffer can compensate for short, isolated interruptions - or a drop in writeout bandwidth - of disk writeout operations of up to 3.8 seconds at maximum recording frame rate. Such interruptions can occur, if the operating system has to perform some internal book keeping and management operations, if meta information about the newly written data needs to be updated on the disks, or if the disk drives have to recalibrate their mechanics and electronics to account for changes in operating temperature.

The multi-threaded producer-consumer implementation of the recording process with multiple ring buffers allows for decoupling operations with very strict requirements on processing latency, like communication with the DMA driven capture hardware, from operations with less strict timing requirements, like data writeout to disk.

[7]Some examples of operating systems and filesystems with this limit are e.g., standard Linux version 2.2, FAT filesystems like the ones used on Microsoft DOS, Windows95/98/ME as well as on most USB memory sticks, Microsoft SMB network shares, HFS filesystem on MacOS, AFP Apple file sharing protocol, NFS version 2, and the ISO 9660 filesystem for CD-ROM's.

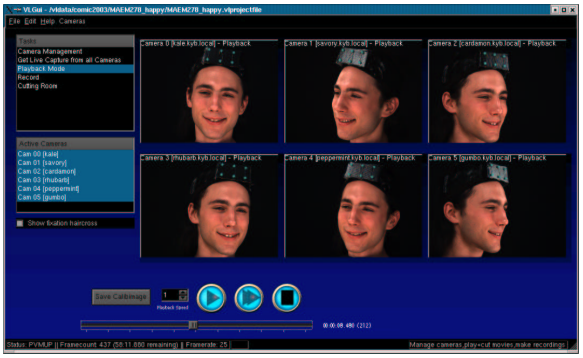

Figure 3: A screen shot of the *vlgui* main application window.

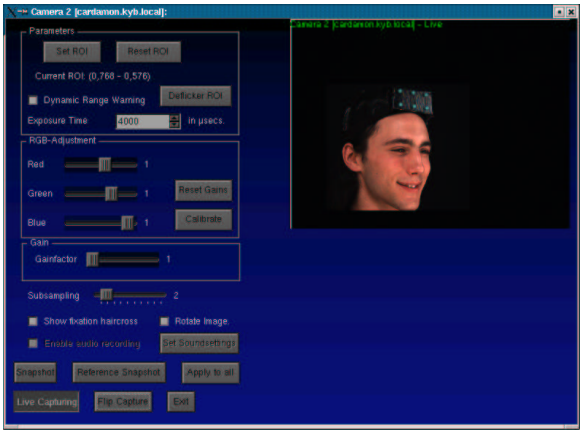

Figure 4: A screen shot of the parameter setup dialog for a single camera. One can see the controls for ROI selection, exposure time, manual and automatic color gain setup, amplifier gain and sound setup, as well as the live preview window.

2.2.4 Application software

This section gives some detail about the applications and tools that are currently available for work with the video setup and its data.

Interactive recording application "vlgui" Our main application for interactive video- and audio recording sessions is called "vlgui". It runs on the VideoLab control computer and provides the human operator with a convenient, easy to use, graphical user interface (GUI) for performing all steps involved in a recording session. Specifically it provides the following features:

- A project setup wizard: This graphical assistant guides the user through all steps involved in preparing the setup for a new series of recordings, like selection of recording frame rate, selection of the subset of cameras used for a recording, and setup of all parameters of all cameras (e.g., region of interest, image format - portrait or landscape, exposure time and signal gain, color calibration, ...). A preview display shows live video from each camera, transmitted over the network, to allow for online change of parameters, adjustment of camera field of view, focus and the like.
  The output of the wizard is a template project - comparable to template documents in a word processor - that contains all the settings for creation of new recording projects by cloning of the template project. The template project does not contain video data.

- Project creation by cloning of previously created project templates.

- Live preview from all cameras by transmission of video over the network. All camera parameters can be customized by use of control dialogs (Figure 4 shows the control dialog for a single camera).

- Control of the actual hard disk video- and audio recording. Inspection, repetition and simple editing of the recorded video tracks (Figure 3 shows the main application window).

- Backup of the recorded projects to the fileserver.

- System management tasks like e.g., deleting old recordings, data backup, system startup and shutdown via network.

- Interactive error handling: In case of operator errors or system malfunctions, the application interactively guides the user through the necessary troubleshooting steps.

The application is written in C++ for Linux. It uses the libvideolabbase - library for project management and control of the cluster hardware and - via use of the VLRecorder class - implements the Master-Process of the distributed recording system during recording sessions, as described in Section 2.2.3.

For implementation of the graphical user interface, we use the *QT-3 library* from the software company Trolltech [12]. This library of C++ classes provides a very powerful and convenient toolkit for the implementation of user interfaces in a platform independent way. It is available freely on Unix/X11 compatible systems like Linux, FreeBSD or MacOS-X for use in

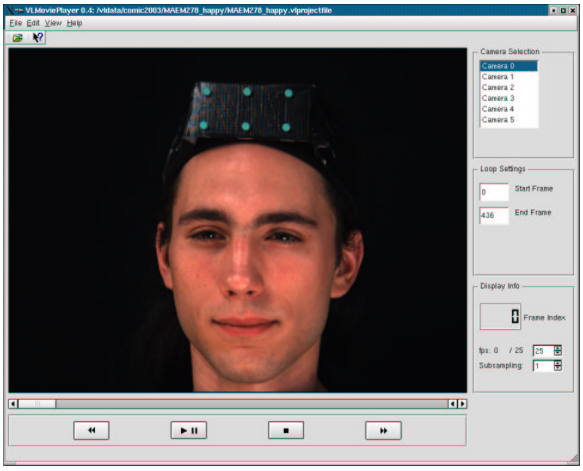

Figure 5: A screen shot of the GUI of *vlmovieplayer*.

conformance with the GPL license[8] [11], and for Microsoft Windows systems via a commercial license.

Videoplayer "vlmovieplayer" As soon as the video data of a recording project has been backed up to the fileserver, the user can use the vlmovieplayer application for reviewing the recorded video content. It currently has the following features:

- Realtime playback of video content with selectable frame rate in 1 fps steps. If the processing power of the display computer is not hight enough, image quality can be reduced via subsampling to allow realtime playback at reduced resolution.
- Frame accurate navigation in the videos and instantaneous switching between the different camera views of a recording.
- Definition of video loops and frame ranges for video editing.
- Manual color calibration and correction.
- Zoom and pan functions for zooming and panning into the video footage.
- Export of video loops and camera views as movie files in standard MPEG-1, MPEG-2 or AVI file formats, that can be played back in standard video players.
- Export of video loops and camera views as sequence of pictures in standard image formats like e.g., Microsoft Bitmap, JPEG, GIF, PNG.

The player is implemented in C++, using the libvideolabbase library for data access and the QT-3 library for the graphical user interface (Figure 5 shows a screen shot). OpenGL is used for fast display, zooming and panning of video. Although it is currently only used on Linux systems, it is portable to other operating systems like MacOS-X or Microsoft Windows, given one has access to the QT-3 library for that systems[9].

Commandline postprocessing tool "vlvideofile2ppm" If one has to perform similar - or identical - editing or postprocessing tasks on a large number of different video recordings, for example , "deflickering" a video sequence (see Section 2.4.2), color correction, image postprocessing or conversion to MPEG movies, it is often more convenient to write a shell script that performs this tasks without human interaction.

For this purpose, we have developed a tool called "vlvideofile2ppm", which is controlled solely via commandline arguments. It provides access to the recorded video- and audio data, implements several frequently needed postprocessing operations on the video footage and exports the processed video data as a stream of bitmap images in the widely portable PPM image format, as well as the audio data in standard WAV file format. The tool can be connected with other flexible postprocessing tools, that are able to take PPM image streams as input, using the powerful concept of Unix command pipelines, to form complex processing workflows on video footage.

The tool is implemented in C, using the libvideolabbase library for data access and is fully portable to other operating systems.

2.3 Performance optimizations

As mentioned in previous sections, realtime hard disk recording of video- and audio data requires two key properties of the recording system as a whole:

- The multi-threaded recording application has to respond to the arrival of new audio- and video data from the capture hardware within a strictly bounded and very short time span, because the hardware supplied DMA memory buffers can store data only for a few milliseconds before it

[8] You are allowed to write applications with no restrictions for internal use. If you want to distribute your application commercially or freely to third parties, you are required to provide the source code of your application and grant all rights specified in the GPL license, including the right to use, modify and redistribute the application and source code under the GPL.

[9] C++, OpenGL and the libvideolabbase (except the VLRecorder class, which is only needed for recording applications) library are fully portable to other operating systems. QT-3 is available for all variants of Unix, including MacOS-X and Windows, but running QT-3 on Windows requires a commercial license.

gets overwritten by more recent data (See Section 2.2.3). This means that only short and isolated response delays of the application can be compensated. Because the synchronization between audio and video is done purely in software by starting the sound capture hardware as fast as possible after arrival of the first video frame, delays in processing the first video frame of a recording would immediately translate into unwanted lag between audio and video content.

- Each recording computer has to consistently write out a datastream of 27 MB/s on average. Only short and isolated reductions in writeout data rate can be compensated by the 100 MB memory data buffer.

This section introduces some measures and modifications to the recording software, operating system and hardware that were done to ensure these constraints are met. These modifications only affect the computers and software of the single recording nodes of the cluster. There is no need to modify the master application, as the control protocol for the recording cluster in combination with the hardware based synchronization between the recording nodes guarantees by design, that the cluster is free of *race conditions*[10], as long as the single recording nodes meet their timing constraints.

2.3.1 Ensuring short processing latencies

The recording work-loop of our application is driven by the hardware timer interrupt of the realtime clock chip (RTC), that is part of all Intel PC compatible computers. Latencies in delivery and handling of hardware interrupts therefore translate into processing latencies of the recording thread. Execution of the critical code paths of our application could get delayed or interrupted by the operating system itself or by other applications running in parallel. We take a couple of measures to prevent this:

- While standard applications and system processes are executed under normal scheduling priority, using the standard Linux timesharing scheduler, the threads of our application are run with *realtime priority*, using the Linux RT_FIFO realtime scheduler. This way, threads of our application preempt other running system processes and applications immediately, as soon as a hardware interrupt is delivered to them. This scheduling mode also guarantees that none of our applications threads can be interrupted in its execution by normal system processes or applications, unless it voluntarily releases the CPU, e.g. to wait for arrival of the next video frame or completion of a disk write request.
  The threads of the recording application also get different realtime priorities assigned, so a higher priority realtime thread can preempt a lower priority one:
  - The *control* thread for handling network RPC requests from the master process has the highest priority, so handling RPC requests from the controlling application always takes precedence over the actual recording work-loop. This is done as a safety and error handling measure to ensure that the recording computers always stay responsive to control commands of the master computer, so that recordings could be stopped at any time by the master application, even in case of a heavy system overload situation due to some hardware fault or software bug.
    This doesn't negatively affect response behavior of the recording workloop, because all RPC handling routines, that could be possibly called while a recording is in progress, are written to ensure bounded and very short processing times.
  - The *grabber* thread has the second highest priority, allowing it to respond as fast as possible to the arrival of new data in the DMA buffers and transfer this data quickly into the big memory buffer.
  - The *writer* thread has lowest priority, because writeout requests could be safely deferred by a couple of hundred milliseconds, as long as this only happens seldomly, while the grabber thread has to respond in time spans of a few milliseconds.
- All pages of the virtual memory of the recording process get *locked* into physical memory, preventing the operating system from paging them out to the systems swap partition in case of memory shortage. If we wouldn't lock process memory, other applications could indirectly impair the realtime behavior of our recording process by allocating memory. This could lead to shortage of physical memory. The operating system could then page out portions of the memory of the recording process onto hard disk storage. The recording process would then get delayed for an

[10] A race condition is a flaw in a system or process where the output exhibits unexpected critical dependence on the relative timing of events. The term originates with the idea that two signals are racing each other to be the first to cause the output.

unpredictable amount of time, possibly hundreds of milliseconds, when trying to access pages of its memory that have been paged out.

- All system applications and services not essential to the recording task (for example, graphical user interface, file- and network servers and system management routines) are disabled[11] during recording sessions, to free up as many processor and memory resources for the recording application as possible.

- To prioritize the handling of network and timer interrupts, that are crucial for quick responses of the *control* and *grabber* threads, over interrupts from the disk subsystem (IDE disk controllers), *interrupt unmasking* is enabled in the Linux IDE disk subsystem: Standard Linux configurations disable the handling and delivery of other higher priority hardware interrupts, while a hardware interrupt signal from the disk controllers is processed. This is done to increase system stability and data integrity, because some combinations of hard disk drives and IDE disk controllers are sensitive to small deviations in timing during handling of disk controller interrupts. Such controllers could malfunction and corrupt data on the disk drives. Fortunately, the IDE controllers and disk drives used in our recording computers work reliably with *interrupt unmasking* enabled.

- Busmaster-DMA (Direct memory access) is enabled on the IDE disk controllers, significantly reducing the workload of the operating system: To write out the content of a data buffer onto hard disk, the operating system only needs to set up the disk controller to perform the task and then start the write operation. The write operation is done automatically by the disk controller, while the main processor can perform other work in parallel, for example, executing the *grabber* thread of our application. Without DMA, the disk driver of the operating system would have to manually drive data transfers to the hard disk, preempting our applications threads from working in parallel and therefore significantly increasing response latencies of the *controller* and *grabber* threads.

This combination of measures leads to a good response behavior of our application. Typical response latencies to timer interrupts are less than 1 millisecond. Processing of a single frame of combined video and audio data by the *grabber* thread normally happens in less than 8 milliseconds, with an average duration of approximately 6 milliseconds, which is sufficient by a safe margin, as the video- and audio DMA buffers could compensate for isolated processing delays of up to 128 milliseconds. The operations of the *writer* thread are usually performed in less than 6 milliseconds.

[11] This is implemented by switching from the normal Unix multiuser runlevel to a special runlevel, that defines the minimum set of services needed to run our recording application.

2.3.2 Improving disk write throughput

The following measures ensure a consistently high disk write throughput:

- We use the Linux Ext2 filesystem for data storage instead of the Ext3 journalling filesystem. While Ext3 provides a very high level of data protection in case of a system crash, and shorter recovery times[12], it would implicate a much higher overhead for managing and writing the internal journal file. In two years of production use we have never had any system crash, and all data is backed up to our fileserver with its data protection mechanisms after each recording session, so special protection against crashes isn't a big priority. Therefore we use the faster, but less protective Ext2 filesystem[13].

- The IDE controllers of our hard disks are finetuned for maximum throughput. We benchmarked different adjustable settings of the Linux disk subsystem to find the settings that give maximum write throughput with minimal processing overhead by the operating system. Enabling Busmaster-DMA in UltraDMA-Mode 2 with unmasked interrupts, 32 Bit data transfers and multiple sector transfers per interrupt proved to be most effective. This is enabled on both IDE controllers with the following command, that is executed during system bootup:
  hdparm -d1X66 -c3 -m16 -u1 /dev/hda

- Tuning the IDE controllers and using Ext2 as filesystem allows data writeout rates of up to 31 MB/s, but only when the disk drive is nearly empty and therefore data is written to files on the outermost tracks of the hard disk.
  The reason for this is the way that hard drives are organized: Hard disks are "filled up" during write operations starting at the outer tracks and proceeding to the inner tracks. Outer tracks on

[12] Filesystem checks on our hard disks take about 20 minutes with Ext2. A journalling filesystem usually takes only a few seconds for filesystems of the same size.

[13] On the few occasions where we had system crashes during the development phase, the Ext2 filesystem was always able to repair itself without data loss during reboot, at the cost of annoyingly long check times.

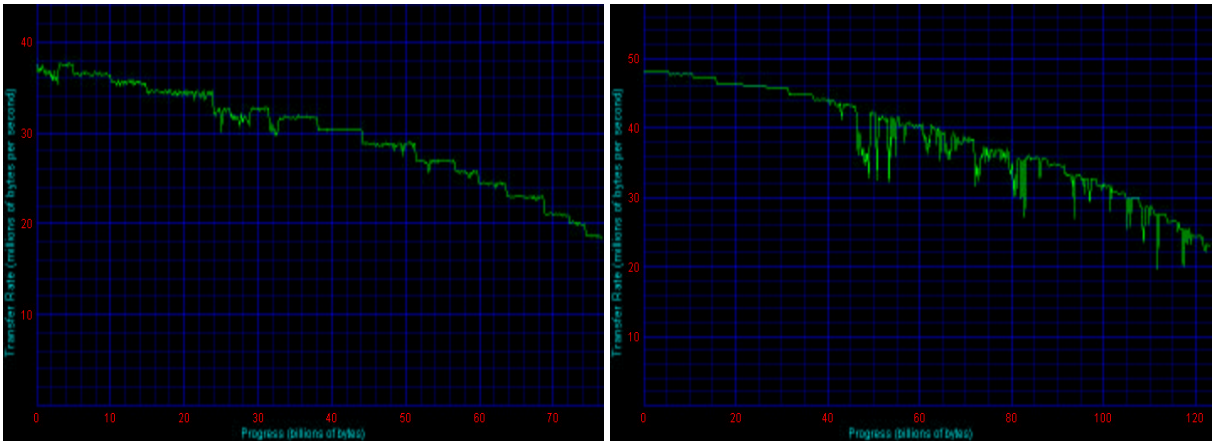

Figure 6: Linear write data rates of the IBM DTLA-307075 disk drive (left figure) and of the IBM IC35L120AVVA07 drive (right figure), plotted as functions of written amount of data. Plots and benchmarks from [13].

the disk have a bigger circumference and therefore a bigger storage capacity than inner tracks. Because disk drive platters rotate at a fixed angular velocity, an outer track, which contains more data, is written in the same amount of time as an inner track, which contains less data, leading to a higher effective write- and read data rate. Therefore, write data rate of a drive is a monotonically decreasing function of write position on the drive - and therefore a monotonically decreasing function of increasing fill-level. Figure 6 shows plots of data rates versus occupied capacities for our models of disk drives (benchmarks and plots from Storageview [13]), nicely illustrating the drop in write bandwidth as a function of disk fill level.

To utilize the full capacity of our disk drives by ensuring a sufficiently high write data rate over the full capacity, we installed two IDE drives in each computer. We use the Linux software RAID subsystem to combine these two physical drives into a RAID level 0 ("Striping configuration") virtual disk drive: Data written out to this virtual drive is transparently split up by Linux into chunks of 64 KB size. Successive chunks are written to the drives in parallel in an alternating fashion, e.g., while the i-th chunk is written to drive 1, the i+1-th chunk is written at the same time to drive 2. This way, the virtual RAID-0 drive achieves a storage capacity and write data rate that is the sum of the capacities and write data rates of the two physical drives. For this to work, it is important that each drive is connected to its own dedicated IDE controller, so that the drives can handle parallel read and write requests without competing with each other for IDE channel bandwidth. Our striping RAID allows the virtual hard disk to achieve a sustained minimum linear write datarate of approximately 39 MB/s, safely exceeding the required minimum average video write data rate of 27 MB/s that is needed by our application, although each physical drive has an average sustained writeout datarate of only approximately 20 MB/s when it is nearly full.

2.4 Lighting

The lighting of our video studio also needs to meet specific requirements. We want the lights to be freely controllable in intensity over a wide range and we want them to be freely moveable and adjustable in orientation, so that we are able to select the best tradeoff between selected light intensity and exposure time for a given scene and the most suitable configuration of lights. During most recording sessions, especially during facial recordings, we want to obtain views of the target subject that are bright enough to allow for the short exposure times necessary for blur-free recording of facial motion, while avoiding strong shadows, specular highlights or non-diffuse light distributions. The scene should also have a natural appearance - colors should look as expected under daylight illumination.

2.4.1 Lighting equipment

Currently we have 4 studio lights, model HD20 of the German company "Hedler-Systemlicht" [14]. Each light is equipped with two light bulbs of a maximum power of 1000 W. These can be switched on and off separately and they are freely dimmable, allowing us to select light emission intensities between zero and up to 2000 W at a color temperature between 1800 Kelvin and 3200 Kelvin, depending on selected light intensity. The lights can be left on for long periods, because they are equipped with cooling fans. The lights have a flexible mounting that allows them to be

oriented freely. They are attached to poles, so they can be moved easily to different locations and heights, giving us the necessary freedom in selection of light arrangement for specific shots.

To handle the heat emissions of such powerful lights, the studio is equipped with a powerful air-conditioning unit.

2.4.2 Removing image flicker

The lights of our video studio are powered by regular alternating current with a frequency of 50 Hz, because electric rectifiers that can convert alternating current into directed current with an output power of 2000 W would have been prohibitively expensive for us to install and maintain. As a consequence, the light sources do not output a constant light intensity, but the light intensity is sinusoidally modulated with a frequency of 50 Hz. Therefore, the mean luminance of the scene changes during the exposure interval of each video frame.

If the selected recording frame rate divides evenly into 50 Hz, e.g., 50fps, 25fps, 10fps, 5fps, then the same portion of each period of the sinusoidal intensity curve is integrated over the exposure interval. Therefore the recorded mean scene luminance stays the same for each captured frame, so the recorded video is free of any flicker.

Temporal aliasing problems arise if the recording frame rate doesn't evenly divide 50 Hz, e.g., at our maximum frame rate of 60 fps. Now, a 50 Hz sinusoidal light modulation curve is "sampled" by a 60 Hz exposure function: Each consecutive video frame integrates luminance over a different part of the 50 Hz light intensity curve, leading to a different mean luminance in each recorded video frame. This results in strong image flicker.

As we want to be able to record at frame rates as high as 60 fps without image flicker, we apply the following procedure as a postprocessing step to the recorded video footage, in order to "measure" the flicker and compensate for it:

1. We select a rectangular subregion R in one of the views of one of the recorded video streams. This image region needs to contain a part of the scene that doesn't change its appearance due to dynamically changing scene content, occlusion or shadows. The 50 Hz light intensity modulation has to be the only source of change in pixel intensity values of the region R. A typical choice for R would be a portion of the static scene background. The selection of an appropriate region is done interactively in the user interface of our recording application *vlgui*.

2. Our postprocessing algorithm calculates the mean over the intensity values of all image pixels contained in R for each frame of the recording: Let $I_t(x, y)$ be the intensity value of the pixel at position (x, y) of video frame t. Then we compute the mean intensity $\tilde{I}_t$ of frame t as:

$$\tilde{I}_t = \frac{\sum_{(x,y)\in R} I_t(x, y)}{|R|}$$

   As R is supposed to be a part of the constant scene background, any change of values in the series $\tilde{I}_1, \tilde{I}_2, \ldots$ is caused by the sampling of the 50 Hz light intensity modulations and is therefore a measurement of image flicker over time[14].

3. The mean over all intensity measurements of a recording with n frames $\tilde{I} = \frac{1}{n}\sum_{i=1}^{n} \tilde{I}_i$ is computed from the per-frame measurements $\tilde{I}_t$. In a flicker-free recording, $\tilde{I} = \tilde{I}_t \; \forall t = 1, \ldots, n$ would hold. Therefore we choose the mean over the whole sequence $\tilde{I}$ as reference intensity and compute a per-frame intensity correction factor f_t for each frame of the recording via:

$$f_t = \frac{\tilde{I}}{\tilde{I}_t} \; \forall t = 1, \ldots, n$$

4. The time series f_t of correction factors is computed once for each recording project as an offline processing step by our commandline batch-processing tool *vlvideofile2ppm* and stored as part of the project file structure on the fileserver. This time series can now be used for on-the-fly flicker correction: Whenever a video frame with an index t of the video recording is accessed from some application by a call to the *getFrame()* method of the *VLVideofile* class (see Section 2.2.1), the corresponding flicker correction factor f_t is multiplied to *all* color components of *all* pixels of the video image, assuming that the measurement of mean intensity modulation $\tilde{I}_t$ of the reference region R in each image is representative for the intensity modulation of the whole image:

$$I_{t,new}(x, y) = f_t I_{t,old}(x, y) \; \forall (x, y)$$

If the reference region R for a specific scene is chosen properly, this on-the-fly correction effectively eliminates image flicker. More powerful correction algorithms could be used if needed, as the original raw data never gets changed, but the deflicker procedure is only applied during readout of video frames from the video files.

[14] This assumes that sensor noise is independent of time.

2.4.3 Color correction

Color correction of the recorded video material is another important post processing step. Since they use halogen bulbs, our studio lights emit light with significantly more power in the "red" (long wavelength) part of the spectrum than in the "blue" (short wavelength) part of the spectrum. This gives all recorded footage a reddish appearance. To reduce this effect, we mount daylight filters in front of all lights to reduce part of the light emission in the "red" part of the spectrum. Each camera has an integrated Infrared-Cutoff filter to block out incident light in the infrared part of the spectrum.

Additionally, we apply the following linear color correction algorithm to the video footage to remove remaining reddishness:

1. A white uniform sheet of paper is placed into the field of view of each camera and a short calibration video recording is made of this white paper.

2. A rectangular subregion R of the recorded video image, which only contains the white paper, is selected by the user in the *vlgui* recording application.

3. The calibration algorithm computes the mean intensity over the red, green and blue color components of all n pixels in region R over all k recorded images:

$$\tilde{I}_R = \frac{1}{kn} \sum_{t=1}^{k} \sum_{(x,y)\in R} I_{t,R}(x,y)$$

$$\tilde{I}_G = \frac{1}{kn} \sum_{t=1}^{k} \sum_{(x,y)\in R} I_{t,G}(x,y)$$

$$\tilde{I}_B = \frac{1}{kn} \sum_{t=1}^{k} \sum_{(x,y)\in R} I_{t,B}(x,y)$$

4. The region R - by selection - only contains pixels that should appear "white", therefore a video recording with perfectly color calibrated cameras should fulfill the constraint: $\tilde{I}_R = \tilde{I}_G = \tilde{I}_B = I_{Ref}$. We define $I_{Ref} = \frac{\tilde{I}_R + \tilde{I}_G + \tilde{I}_B}{3}$ and compute color correction gain factors for each color component to equalize the mean red, green and blue intensity to the reference value I_{Ref}:

$$f_R = \frac{I_{Ref}}{\tilde{I}_R}$$

$$f_G = \frac{I_{Ref}}{\tilde{I}_G}$$

$$f_B = \frac{I_{Ref}}{\tilde{I}_B}$$

5. These color gain factors are stored as part of the project configuration files of all recordings made with the given light and camera setup. They are applied on-the-fly to images, by the *getFrame()* method of the *VLVideofile* class, whenever an application requests a specific image of a recorded video sequence. Each color component of each pixel of the RGB image is multiplied by the corresponding gain factor to produce the color corrected output image:

$$I_{out,R}(x,y) = f_R I_{in,R}(x,y) \; \forall (x,y)$$

$$I_{out,G}(x,y) = f_G I_{in,G}(x,y) \; \forall (x,y)$$

$$I_{out,B}(x,y) = f_B I_{in,B}(x,y) \; \forall (x,y)$$

Although this algorithm can only perform a simple linear color correction, it has been sufficient for our purposes up to now. As the recorded raw video data is never touched, but the color correction is applied as an on-the-fly postprocessing step during readout of video data, more advanced algorithms could be applied any time later after recording.

2.5 Lenses

Our cameras can be used with any type of lenses that have a standard C-Mount adapter for connecting to the camera and that are useable with a 1/2 inch CCD sensor. Currently we have two sets of lenses:

- For facial recordings, we use narrow angle lenses, model "Xenoplan" from the German company "Schneider Kreuznach Optik", with a fixed focal length of 23mm and an aperture range of 1.4 to 11. They are well suited for sharp facial recordings at a distance between face and cameras of approximately 1.5 m.

- For full body recordings of human actions, we have wide angle lenses, model C60402 from "Cosmicar/Pentax" with a fixed focal length of 4.2 mm and an aperture range of 1.6 to 16.

3 Conclusions

In this report we have described the technical details of the MPI VideoLab as a high performance synchronized, multiple camera recording setup. Instead of a detailed summary, we rather want to provide a short list of references to projects that have been successfully completed since the VideoLab became operational in May 2002.

Up to now, video recordings in the VideoLab were done on human facial expressions, head gestures and simple actions. While some of the video material was used "as is" for presentation to human subjects as part

of experiments on face and action perception (see e.g., [16] and [17]), some experiments required the application of computer vision and computer graphics algorithms to the video footage as a postprocessing step. Typical examples are the removal of head movements from video footage, video based manipulation of facial expressions and video based manipulation of parts of the face (see e.g., [18] and [19]). These postprocessing steps involve algorithms for high precision 3D tracking of head movements (e.g., [20]) and for video based texture extraction and manipulation.

This project was funded by the Max Planck Society. We would like to thank Ian Thornton for initially proposing this project and Barbara Knappmeyer for helping to design the studio lighting. We would also like to thank Ian Thornton, Quoc Vuong and Martin Breidt for useful comments on this technical report.

References

[1] Basler Vision Technologies: `http://www.baslerweb.com`

[2] Jensick, J., "Dueling Detectors", *oemagazine - The monthly publication of The International Society for Optical Engineering, Vol. 2-2*, February 2002.

[3] Silicon Software: `http://www.silicon-software.de`

[4] B. E. Bayer, "Color imaging array", *United States Patent 3,971,065*, 1976.

[5] R. Ramanath, W. Snyder, G. Bilbro and W. Sander, "Demosaicking methods for bayer color arrays", *Journal of Electronic Imaging, Vol 11-3, pp. 306-315*, July 2002.

[6] Linux online: `http://www.linux.org`

[7] FSMLabs: `http://www.fsmlabs.com`

[8] Homepage of CPPvm: `http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm/`

[9] Homepage of PVM: `http://www.csm.ornl.gov/pvm/`

[10] LGPL license text: `http://www.gnu.org/copyleft/lesser.html`

[11] GPL license text: `http://www.gnu.org/licenses/gpl.html`

[12] Homepage of Trolltech, provider of the QT library: `http://www.trolltech.com`

[13] Homepage of Storageview: `http://storagereview.com`

[14] Hedler Systemlicht: `http://www.hedler.de`

[15] Homepage of Samba, the free SMB fileserver software: `http://www.samba.org`

[16] Cunningham, D.W., M. Breidt, M. Kleiner, C. Wallraven and H.H. Bülthoff, "How Believable are Real Faces: Towards a Perceptual Basis for Conversational Animation", *Proceedings of the 16th International Conference on Computer Animation and Social Agents, IEEE Computer Society, ISBN 0-7695-1934-2*, 2003.

[17] Pilz, K., I.M. Thornton and H.H. Bülthoff, "Matching and Searching for Moving Faces", *Third annual meeting of the Vision Sciences Society, Sarasota (Florida), Journal of Vision, Vol 3-9, ISSN 1534-7362*, 2003.

[18] Cunningham, D.W., M. Breidt, M. Kleiner, C. Wallraven and H.H. Bülthoff, "The inaccuracy and insincerity of real faces", *Proceedings of Visualization, Imaging, and Image Processing 2003, Vol 1, ACTA Press, ISBN 0-88986-382-2*, 2003.

[19] Schwaninger, A., D.W. Cunningham and M. Kleiner, "Moving the Thatcher Illusion", *10th Annual Workshop on Object Perception and Memory 2002, Psychonomic Society Publications*, 2002.

[20] Kleiner, M., "Ein stereobasiertes Verfahren zum dreidimensionalen Tracking von Markern in menschlichen Gesichtern", *Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Lehrstuhl für Graphisch-Interaktive Systeme, Universität Tübingen*, 2001.