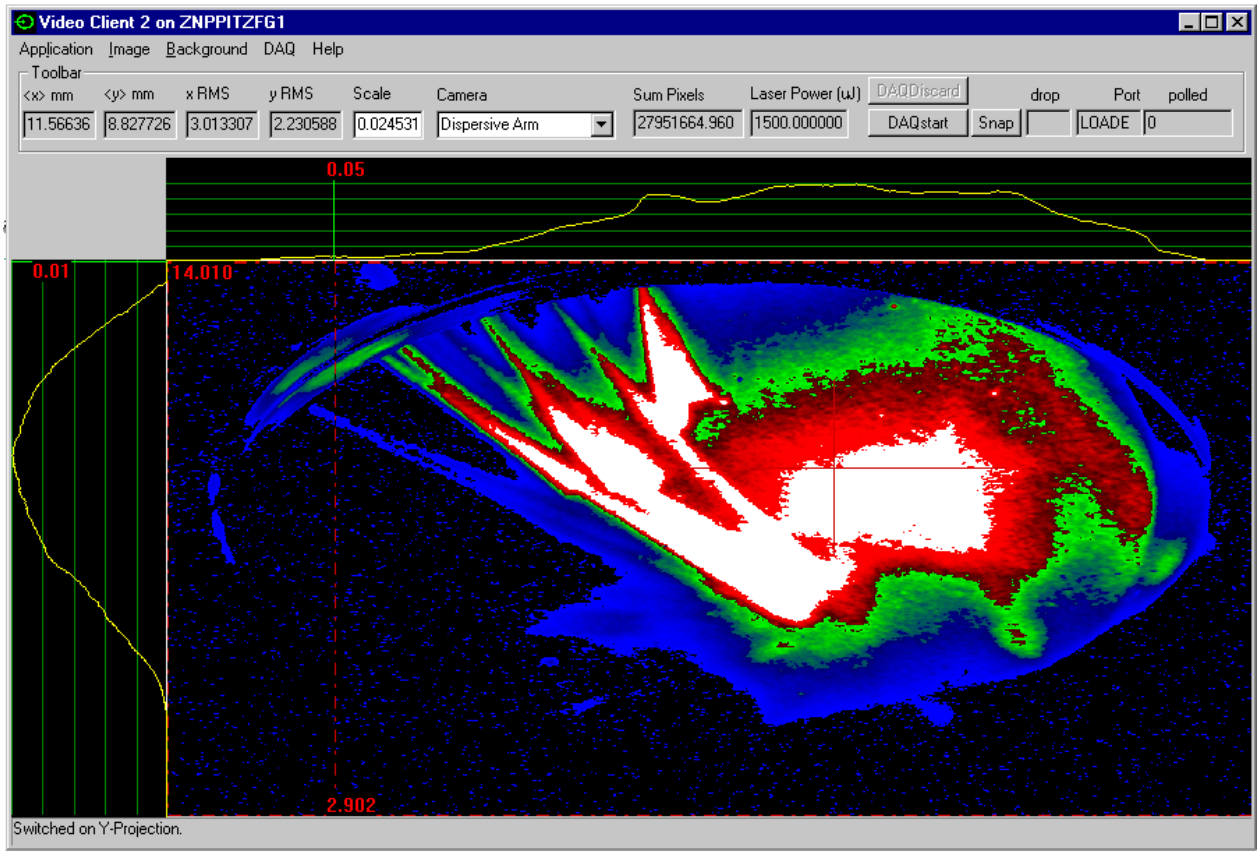
Video Client 2

A Program to Analyse Video Frames

Programmer's Documentation



by Stefan Weiße

sweisse@ifh.de

March 18, 2003

Table of Contents

1. Introduction

This program is meant to analyse spots on video frames. It is needed in the PITZ experiment, because the optimisation of an electron gun is only possible based on an extended diagnostic system including a Video-system. The goal is to measure the electron beam position and the profile of the beam at different places and by different diagnostic tools along the beam line.

Using this program one is able to continuously receive frames ('Poll Mode') or acquire a certain amount of frames ('Grab Mode') from a server. It is possible to apply some analysis on the frames like X-ray filtering, background subtraction, normalization, spot centre and size detection based on multiple algorithms, false colour mode etc. One is able to load and save frames as well as background images using certain file formats.

Due to heavy optimisation techniques the program is able to apply most calculations in real-time. On a Pentium III 1000 MHz one is able to watch the video 'with all eye candy on' at 10 Hz. MMX instructions and special instructions of Pentium II and faster processors were used. So at least a Pentium II processor is required. Recommended is a Pentium III 866 MHz or faster processor. On a Pentium IV PC >= 2 GHz it is possible to run two instances simultaneously (with all eye candy on) watching at two different cameras at 10 Hz.

The frames come in either by a socket connection or via TINE protocol in multicast operation mode. The frames are compressed. A loss-less real-time compression with a compression ratio of about 2:1 to 3:1 is used. Sending uncompressed frames over the network would eat up the bandwidth of a 100 MBit network. A single frame is made up out of greyscales with a width of 768 pixels and a height of 574 pixels. Sometimes frames are rotated so width and height can be swapped.

Apart from the socket connection to receive the frames, it is possible to control the server. One can switch the camera port from which frames are grabbed. A server can have up to 4 cameras connected to it, but from only one camera can be grabbed at a given time. This control connection is implemented using the TINE protocol. The TINE protocol is also used to transfer camera port description strings, scale values, width and height of certain camera ports etc. from server to client.

Key features of the client:
- multi-threaded
- real-time decompression of loss-less compressed images
- highly optimised algorithms (all of them can be applied in real-time)
- switching of the camera port
- spot size and centre calculation using straightforward or Fourier analysis
- background subtraction, X-Ray filtering, normalization, false colour mode
- selectable area of interest
- projections (X and Y) of the video signal
- driven by configuration file

The client is written in C++ language, by making use of the integrated development environment Microsoft Visual C++ 6. Inline assembler is used in some speed critical parts of the program. The

source code is about 20000 lines long (660 KB) including comments. It is very well inline commented.

Information on how to use the software is collated in the *Video Client User Documentation*. This documentation is meant only for programmers, who want to understand how the software works.

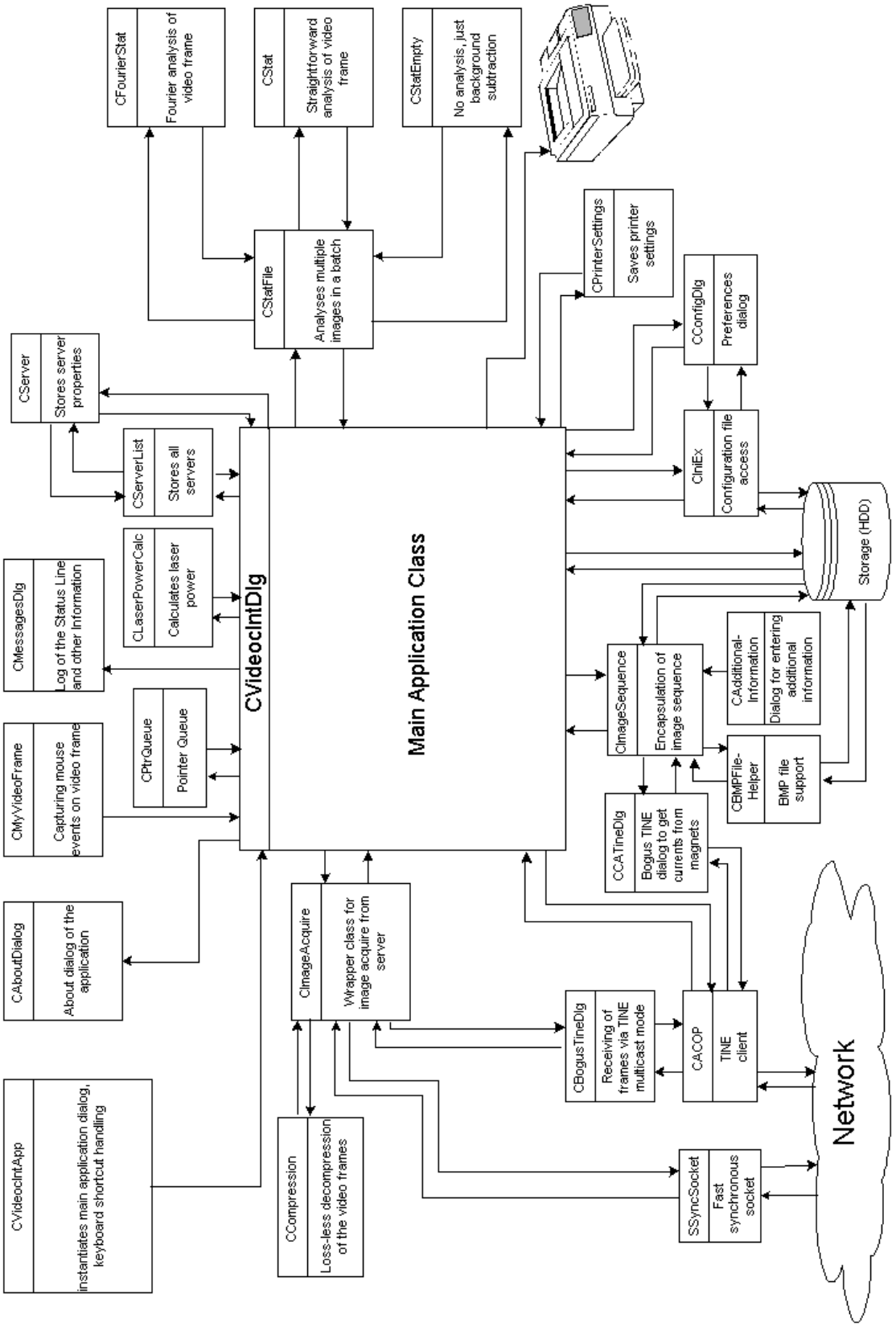
2. Overview of the Classes

The program is constructed out of 26 classes. Most classes are helper classes for the main class of the application, which is the CVideoclntDlg class. The CVideoclntDlg class is the class behind the video client's main dialog and manages the whole program.

At first, the classes are enumerated and a short description what each class does is given.

Class	**Description**
CAboutDialog	about dialog of the application
CACOP	wrapper class of the ACOP ActiveX control
CAdditionalInformation	dialog class to enter additional information when saving files
CBMPFileHelper	helper class for BMP file support
CBogusTineDlg	bogus dialog for receiving frames using TINE protocol
CCATineDlg	bogus dialog for receiving magnet current information
CCompression	decompression of the compressed video frames
CConfigDlg	configuration dialog class
CFourierStat	spot centre and size detection based on Fourier analysis of the frames
CImageAcquire	wrapper class for image acquire from server
CImageSequence	class to hold the image/image sequence and additional properties
CIniEx	class for accessing the initialisation file
CLaserPowerCalc	loading of lookup-table and calculation of laser power based on LUT
CMemoryManagement	aligned memory allocation and wrapper for memory copy
CMessagesDlg	window class for the messages dialog (extended information and log)
CMyVideoFrame	class for capturing mouse events on the video frame (onto the dialog)
CPrinterSettings	class which saves the settings from printer dialog for next showing
CPtrQueue	a pointer queue for transmitting frames across thread boundaries
CServer	encapsulates all properties for a single server
CServerList	list to store all connected servers
CStat	base class for spot centre and size detection, needed by CFourierStat
CStatEmpty	CStat without statistical analysis, needed when statistics are off
CStatFile	wrapper class around the statistical classes
CVideoclntApp	application class, just instantiates the main dialog
CVideoclntDlg	main class of the video client, manages the whole application
SSyncSocket	fast synchronous socket class

2.1. Class Chart



2.2. Description of the Classes

CAboutDialog **about dialog of the application**

This class is a dialog resource wrapper class in order to display the about dialog of the application. It is used in CVideoclntDlg class.

CACOP **wrapper class of the ACOP ActiveX control**

This class was entirely generated by Visual C++. It is a wrapper class in order to properly access the 'ACOP' ActiveX control, which is a TINE client. Using this wrapper class the control connections, like changing the camera port and requesting and receiving the camera strings from the server were implemented.

This class is used in CVideoclntDlg class.

CAdditionalInformation **window class to enter additional information (saving files)**

This is the wrapper class for the 'Additional Information' dialog resource ('IDD_DIALOG1'), which pops up when saving files or backgrounds. It consists of some wrapper code as well as a lot of call-back procedures which react to certain events.

This class is used in CImageSequence class.

CBMPFileHelper **BMP file access functions**

This class contains useful helper functions when working with BMP files. Using this class the application is able to read and write BMP files. It is used through CImageSequence class. It is also used in CVideoclntDlg for saving and loading backgrounds in BMP format.

CBogusTineDlg **bogus dialog for accessing a TINE control for receiving frames**

This class creates a bogus dialog in order to access the TINE ActiveX control for receiving frames using TINE multicast. It is used in CImageAcquire class.

CCATineDlg **bogus dialog for accessing a TINE control for magnet properties**

This class creates a bogus dialog in order to be able to retrieve magnet current information for main solenoid and bucking coil. It is used in CImageSequence class.

CCompression **decompression of the compressed video frames**

This class is needed for decompression of the video frames. It encapsulates all functionality that is needed to decompress the video frames that come in through the socket connection or via TINE protocol. This class is used in CImageAcquire and CBogusTineDlg class.

CConfigDlg **configuration dialog class**

This is the MFC class for the configuration dialog. It is connected tightly to the resource of the configuration dialog. By using the dialog the user is able to change the preferences of the application. The required preferences are loaded from and saved to a configuration file by use of the CIniEx class. There are some functions that get activated when the user makes certain operations on the dialog's controls ('On…'-functions).

Using the configuration dialog, one can force reinitialisation of server settings, select snapshot directory, repetition rate and transfer mode. This class is used in CVideoclntDlg class.

CFourierStat **spot centre and size detection based on Fourier analysis**

This class inherits the CStat class. It replaces the spot centre and size calculation . These values are calculated based on Fourier analysis. This class is needed for proper calculation of spot centre and size. In addition to the spot centre and size detection of the spot the class also does background subtraction, normalization and calculation of the projections.

This class is used in CStatFile class.

CImageAcquire **encapsulation of acquiring images from the server**

Using the CImageAcquire class, the application is able to acquire frames from the server using poll or grab mode. The transfer mode is autodetected and the frames are also decompressed here. The class supports TINE multicast mode through access to CBogusTineDlg and pure streaming socket connections. This class is used in CVideoclntDlg class.

CImageSequence **encapsulation of images / image sequences and image properties**

The class CImageSequence encapsulates the access of frame buffers and holds all properties that describe a frame. Inside the class, the DOOCS and Channel Access properties that can be stored together with a video frame file are acquired when necessary. In addition, raw image files can be loaded and saved. This class is used inside CVideoclntDlg class.

CIniEx **class for accessing the initialisation file**

This class is needed for initialisation file handling. It is an external class, which was downloaded from the web. One can load and save an initialisation file and access the members of the file through a Section/Key/Value schema.

This class is used in CVideoclntDlg and CConfigDlg class.

CLaserPowerCalc **loading of lookup table and calculation of laser power in µJ**

Using the CLaserPowerCalc class it is possible to calculate the Laser power of the spot that is seen on a video frame. In addition, a lookup-table that holds the calibrated laser power can be loaded in.

CMemoryManagement **aligned memory allocation and wrapper for memcpy()**

The CMemoryManagement class is a wrapper class for aligned memory allocation and memcpy(). One can easily add different implementations of aligned memory allocation and memory copy routine(s).

This class is global to the whole application.

CMessagesDlg **window class for the messages dialog (log)**

This is a class for handling a separate messages window. It is useful for debugging. In addition, every status line which is displayed is logged onto the messages window.

This class is used in CVideoclntDlg class.

CMyVideoFrame **class for capturing mouse events on the video frame**

This class is needed to capture the mouse events that are happening on the video frame. This is done by inheriting from CStatic class and extending the class by the needed functionality.

This class is used in CVideoclntDlg class.

CPrinterSettings **stores current printer settings**

The class CPrinterSettings is needed to store the current printer settings for subsequent calls of the printer dialog. This is an external class and was downloaded from the web.

This class is used in CVideoclntDlg class.

CPtrQueue **a pointer queue for transmitting frames**

This class was written to transport grabbed frames across thread boundaries. One inputs the pointer to the frame on one side, and on the other side it is possible to get the pointer out of the queue. The whole class is thread-safe at object level. This is achieved by using the MFC synchronisation classes CSingleLock and CCriticalSection.

This class is used in CVideoclntDlg, CBogusTineDlg and CImageAcquire class.

CServer **encapsulation of server properties**

The class CServer is used for storing settings that belong to a server. A server is the hard-ware/program combination from which frames are aquired. A server has settings like TINE Eqpname, listen port, hostname, scale values, camera port description strings, width and height of images and bits per pixel setting. This class is used inside CServerList and CVideoclntDlg.

CServerList **linked list of all installed servers**

This class stores the list of servers that can be used in the client. One is able to add and delete servers and get the desired server class out of the list. This class is used inside CVideoclntDlg class.

CStat **base class for spot centre and size detection (straightforward)**

This class is the base class for statistical analysis of the video data. It features spot centre and size calculation based on 'Weight of the Masses' algorithm. In addition to this, the class also features background subtraction, normalization, calculation of the projections and X-ray filtering.

This class is used in CStatFile class.

CStatEmpty **CStat without statistical analysis, needed when statistics are off**

This class is like a sister class of CStat. It is needed when statistics are switched off. Nearly no calculation/analysis takes place. Only the background is subtracted and the image is maybe X-Ray filtered. From the public members point of view, the class looks the same as CStat class.

This class is used in CStatFile class.

CStatFile **wrapper class around the statistical classes**

This class does the statistical analysis. It makes use of CStatEmpty, CStat and CFourierStat clas-ses to support no, straightforward and fourier statistics. It is kind of an extension to the other stat. classes to support analysis of multiple images in a batch. When multiple images are analysed in a batch, the results are averaged (mean values) over all analysed images.

This class is used in CVideoclntDlg class.

CVideoclntApp **application class, just instantiates the main dialog**

This is the main application class. The execution path starts here. It is responsible for properly starting up the main application dialog (CVideoclntDlg). All functionality is then handled there.

CVideoclntDlg **main class of the video client, manages the whole application**

This is the main class of the application. All other classes are just helper classes for this main class. Most events are received here and processed accordingly. There are a lot of 'On…' func-tions which react on the menu entries, hotkeys and controls.

SSyncSocket **fast synchronous socket class**

The class SSyncSocket provides socket functions to the application. It encapsulates the raw Windows socket API. It is synchronous, which means that each operation, which can not be processed immediately, blocks the execution flow. Because of this, most of the functions are only called from threads, in order not to block the main execution flow.

This class is used in CVideoclntDlg class.

2.3. Application Flow

In this section, a short overview of the execution flow of the application is presented. When the program is started, the InitInstance() procedure in CVideoclntApp class instantiates the main window of the application (CVideoclntDlg class). Then the main execution flow starts at OnInitDialog() function in CVideoclntDlg class. Inside this function, the main initialisation takes place. All member variables are set to their default value, the memory locations for storing grabbed frames, backgrounds, normalized images etc. are allocated. The initialisation files are read in or created, if the files could not be found. The camera names of all servers together with the camera port scales and other important information (width, height and bits per pixel setting of video frames) are downloaded from the servers specified in the config files using TINE protocol. In addition, the decompression and drawing subsystem of CVideoclntDlg are initialised. If something terribly happens, the program displays an error message and the whole application quits.

If all initialisation goes well, the program starts listening for user actions like pressing buttons or hotkeys, selecting menu entries or moving the mouse. The flow of some special functionality of the program will now be presented.

Poll Mode

When poll mode is switched on by hotkey or via the menu, the procedure StartPollMode() of CVideoclntDlg() is called. This function calls another function StartPollMode() inside CImageAcquire class. This function checks whether TINE protocol or streaming sockets should be used to transfer the video frames from server to client. If TINE protocol is available, another class CBogusTineDlg is called and the transfer using TINE protocol is started. For streaming socket mode, the CImageAcquire class has all functionality to acquire the frames over the socket connection built in. If socket mode is selected, the socket is instantiated and connected to the server. If everything goes well, the receive thread is started.

For socket mode, the frames are acquired using a thread. In TINE mode, it is a bit different. An event function is called for every new frame that comes in. Both routines first decompress the frame and put the frame into the receive queue (CPtrQueue) afterwards. In addition, for every frame there is a Windows message posted, which results in the main thread that the function OnFrameReceived() is called.

In this OnFrameReceived() function, the current chosen statistical analysis is done on this frame. This is done through the CStatFile class, which chose the proper statistical analysis class. Inside the proper analysis class, the chosen statistics are calculated, the background is subtracted and the area of interest is possibly normalized, the projections are calculated etc.
After the statistical analysis, the processed frame is displayed using DisplayFunction() method of CVideoclntDlg class. Inside this function, the frame is drawn (using Windows DrawDib() API) and stored as the last frame (if it needs to be redrawn). In addition, the statistical values, the projection area (if necessary), the mouse cross, the red frame around the area of interest are drawn. The DisplayFunction() is a complete redraw of the application window. After a successfull call to the DisplayFunction(), some values regarding to poll mode are updated on the toolbar.

The last two paragraphs are traversed for each new frame that comes in.

When the user wants to switch of poll mode, the function StopPollMode() is started. This function calls the function StopPollMode() inside CImageAcquire. Using this function, either the receive thread is shut down (for socket mode) or the link to the server (TINE mode) is closed. After this, some variables will be cleaned up for the next start of poll mode. This is everything that needs to be done. By shutting down the receive thread or closing the link to the server one can be sure that the function OnFrameReceived() will not be called again.

Grab Mode

It makes no difference how many frames will be grabbed. For every grabbing frames process, the function GrabFramesFromServer() in class CVideoclntDlg is called. In the parameter list, one specifies how many frames should be grabbed and the width and height of a frame. The function GrabFramesFromServer() inside CVideoclntDlg class calls the function GrabFrames() inside CImageAcquire class. This function decides whether TINE protocol can be used to acquire the frames or socket mode should be used.

For socket mode, all required functionality are built into the CImageAcquire class. At first the socket (SSyncSocket class) is instantiated and connected to the passed server. If the connect is successful, the grab thread is started. Then the function wait for the grab thread to finish or a timeout occurs. All received frames are decompressed and put into a pointer queue (CPtrQueue).

For TINE mode, the CImageAcquire class calls a second class (CBogusTineDlg) which handles the grabbing of frames using TINE protocol. A link to the server is opened and through a notify function the frames are acquired from server, decompressed and put into a pointer queue (CPtrQueue).

During grab mode, some windows messages can be posted to the main application. These are:

ID_MSG_SOERRGRAB	posted when an error happened on grabbing frames
ID_MSG_GRABFINISHED	posted when grabbing has finished without errors
ID_MSG_FRAMEARRIVED	posted for every new frame that arrives on client side

The main application class react on these messages and calls the function of CImageAcquire class in order to properly shut down and clean up grabbing. If any of the first two messages is posted, the function GrabFramesCleanup() of class CImageAcquire is called to properly clean up

after grab mode. After this, the function GrabFramesFromServerCallback() of CVideoclntDlg class is called. This function first checks for the validity of frames. If the frames are valid, they are statistically processed using the CStatFile class. The behaviour is the same as for poll mode. After processing the frame(s), the resulting image is displayed (DisplayFunction()) and all appropriate controls are updated.

Quit Application

When quitting the application, either by pressing the 'X' on the title bar or selecting Application-> Quit from the menu, the function QuitApplication() is called. This function checks whether poll mode is still active and shuts down the poll mode if necessary. After this, the memory blocks (background, normalized image, grab buffer etc.) are freed. Then the drawing, initialisation file and decompression subsystems are shut down. After this, the application is closed.

3. Selected Documentation of Details

3.1. Protocol of the Socket Connection

The protocol between server and client is pretty simple. After the client successfully connects to the server, the server starts to send one video frame after another. Up to now, the server receives nothing through the socket connection.

A full frame is made out of two or three parts. It depends whether the frame is compressed or not. At first, the video header is send out for each frame. This structure is 88 bytes long and contains several important information concerning the frame. For example, the frame number, a timestamp, a compression flag, length of the frame bits and much more useful information is transmitted.

After the video header, the raw data follows. When a frame is send out compressed, a second header (format header) is transmitted after the video header. This header is needed by the codec for properly decompressing the video frame. Because the decompression routines are built into the client, this header is more or less useless. For compatibility reasons, this header is still transmitted. Later, it could be that this header is needed again. Also some old versions of the video client rely on this format header.

After the format header, the frame bits follow. The size of the frame bits is encoded in the video header (VIDHDR.len_data).

If a frame is send out uncompressed, the format header is omitted. Just after the video header the uncompressed frame bits follow. The size of the uncompressed frame bits is also encoded in the video header (VIDHDR.len_data).

An algorithm to receive the frames through a socket connection would be:

- read in the video header (88 bytes)

- is the frame compressed ? (VIDHDR.compressed == 1)
- if yes, read in format header (length: VIDHDR.len_fmthdr)
- if yes, read in compressed frame bits (length: VIDHDR.len_data)
- if not, read in uncompressed frame bits (length: VIDHDR.len_data)
- go to start

When the compressed frame bits are read into a memory location, one should add at least 8 bytes for safety to the memory location. This is needed for properly decompressing the frame bits. To decompress the frames, one can use the CCompression class of the Video Client.

In the following lines, details are explained concerning the video header. Looking at the source, the structure has the following design:

```
typedef struct {
  UINT   kennung;      // 0x11223344;
  UINT   compression;  // FOURCC codec name
  UINT   len_total;    // length of data + formatheader + vidhdr
  UINT   len_data;     // length of the data (compressed image)
  UINT   len_fmthdr;   // length of the format header (needed for
                       // decompression on windows side)
  UINT   frameno;      // since starting the grabserver
  _timeb timestamp;    // time in milliseconds since 1970 // _ftime() _timeb
                       // structure
  INT    cameraport;   // port of the camera (0..3)
  UINT   widthframe;   // width of the frame in pixels
  UINT   heightframe;  // height of the frame in pixels
  UINT   compressed;   // 1= picture is compressed
                       // 0= picture is not compressed because if it would be
                       // compressed it would be too big
  double framerate;    // floating point: average number of frames per second
  double scale;        // floating point: scale value of the video frame
  UINT   bpp;          // bits per pixel settings of the video frame

  char   reserved[12]; // 12 bytes reserved for future enhancements
} VIDHDR;
```

All data is transmitted in Little Endian format (Intel PC). If the data is read in on a different platform (e.g. SUN (Big Endian)), the values must be converted accordingly.

In the following table the structure members are explained.

Name	**Description**
kennung	Identification (marker) of the video header, contains 0x11223344
compression	FOURCC (Four Character Code) of the codec that was used The content would be MAKEFOURCC('H','F','Y','U').
len_total	total length of the video frame (video header + (format header) + frame bits) in bytes
len_data	length of the frame bits in bytes

Name	Description (continued)
len_fmthdr	length of the format header (for a compressed frame) in bytes, for an un-compressed frame this member variable should be zero
frameno	number of the frame (incremented by 1 for each frame)
timestamp	timestamp taken the time the frame was acquired by the framegrabber card
cameraport	the camera port the frame was taken from (0..3)
widthframe heightframe	width and height of the video frame in pixels
compressed	flag that indicates whether a frame is compressed or not 1 = frame was compressed; 0 = frame was not compressed
framerate	floating-point variable with the actual framerate (about)
scale	floating point variable with the scale factor of the video frame
bpp	bits per pixel setting of the video frame
reserved[12]	12 chars reserved for future enhancements

3.2. Protocol of the TINE frame transfer

Aside from the socket connection, there exists another (preferred) possibility to acquire the frames from the server. It is called "TINE multicast mode". Using this mode, the video frames are transferred via multicast using TINE protocol to all connected clients. This saves network bandwidth and server resources. Using the settings obtained from config file, the client knows how to access the frame property using TINE multicast. For each server, there is an entry called "TineEqpName" which is the Device Server in TINE terms. For accessing the server using TINE protocol, 4 parameters are needed. These are:

Device Context: "PITZ"
Device Server: value of TineEqpName
Device Location: "device_0"
Device Property: "FRAME.GET"

Using this Context/Server/Location/Property scheme the client is able to connect to the server and gets the frames. The frames are transferred using TINE Poll mode. The client asks every 50 milliseconds "Server, do you have any new frame?" The server either sends the new frame (if there is any) and returns status code 0. If there is no new frame ready to be send out, the server returns error status code 607. Using these 50 milliseconds the TINE connection is restricted to 20

frames per second. If there should be more frames transmitted, one has to modify the POLL rate to less than 50 milliseconds.

Using TINE protocol, on each receive event one gets a buffer with the current frame data in it. This is only true if the status code that the server has sent back equals to zero. If the status code is not zero, no frame data was transmitted to the client.

The frame data consist of a video header (see section 3.1), one optional frame header (for decompression) and the raw video frame bits (either compressed or uncompressed). One should read out the video header first and check the information contained in the video header. After examining it, one knows whether the frame is valid, compressed or whether it contains an optional frame header. Using this information, one can either decompress the frame or copy the uncompressed frame bits into an appropriate memory location.

3.3. Description of the Configuration File

The client is driven by two configuration files. The first configuration file resides in the same directory as the executable and is called 'videoclnt2.cfg'. It contains the server settings, the default server and the currently set repetition rate of the experiment. The second file resides in the user's profile directory and is called 'videoclnt2_user.cfg'. In it, the snapshot directory is stored. There can be a different snapshot directory for each user that uses the Video Client 2.

Although for most cases it is not necessary to modify the files by hand the entries are explained here. One can adjust some of the settings through the configuration dialog of the Video Client. The server settings can not be adjusted through the configuration dialog. Together with the executable, one gets a default configuration file with preconfigured server settings. This settings should only be changed by experts because filling in wrong values can render the server connections useless.

Both configuration files follow the Section/Key/Value schema. When the client is started, it looks for the configuration files. If no configuration files could be found, the Video Client 2 creates default files.

A example configuration file at the executables directory would be:

---- snip ----
```
[MAIN]
DefaultServer=ZNPPITZFG1
Servers=2
Server001=ZNPPITZFG1
Server002=ZNPPITZFG2
Framerate=0

[ZNPPITZFG1]
Servername=znppitzfg1
Portnumber=31777
TineEqpName=ZNPPITZFG1F

[ZNPPITZFG2]
Servername=znppitzfg2
Portnumber=31777
TineEqpName=ZNPPITZFG2F
```
---- snap ----

At first, there is the the main section. It consists of 4 or more key-value pairs (depending on the number of servers). The entries that are always there are 'Servers', 'DefaultServer' and 'Framerate'. The two keys 'Server001' and 'Server002' are the two installed servers. The number of 'ServerNNN' keys depends on the 'Server' setting above. In this example there are 2 Servers ('Servers=2'). The two keys 'Server001' and 'Server002' specify the section names of the two servers. For every server there exists a dedicated section, which will be explained later.

The 'DefaultServer' key specifies the server section that will be default when starting up Video Client 2. When the client is started up and one starts polling frames the client will connect to this server.

The 'Framerate' key specifies the approximate repetition rate the client expects from the server. The value is of type integer and can range between 0 and 10. 1 would be 1 Hz repetition rate, 10 would be 10 Hz repetition rate. If zero is entered, the Video Client 2 will try to autodetect the repetition rate.

In addition to the main section, there are optional sections which describe the details of the installed servers. The settings are required so that the client knows how to connect to the server. A default server configuration would be:

```
[ZNPPITZFG1]
Servername=znppitzfg1
Portnumber=31777
TineEqpName=ZNPPITZFG1F
```

The Servername is the name of the server. The client uses this information to connect to the server using streaming sockets. It is the DNS name of the grabber server computer. The value is of type string. Please pay attention, the server name is case sensitive.

Together with the Servername, the client needs one more information to successfully connect to the server using streaming sockets. This is the Portnumber. It is an integer value which can range from 0..65535. The default value is 31777.

The third setting is TineEqpName. This is the Device Server in TINE terms. Using this setting the client knows how to get frames using TINE protocol in multicast operation mode. The value is of type string. Under normal conditions the TineEqpName is the Servername setting in upper case plus an 'F'.

The second configuration file, stored in the user's profile directory, looks like this:

--- snip ---

```
[MAIN]
SnapshotDir=h:\
```

--- snap ---

The SnapshotDir key saves the actual snapshot directory the program needs for properly storing the video frame snapshots. The value is of type string. One has to make sure that the directory specified exists and is writable.

3.4. Description of the File Formats of the Video Client

Together with the application, two new file formats were created. The first one is called IMM format and is meant for storing a bunch of unprocessed video frames into a file. The second file format (BKG) is meant for storing background images in a file.

IMM File Format

In a IMM file, a certain number of frames can be saved. All frames go uncompressed into the file. Together with the image data, there are some special variables saved into the file. These are width and height of the frame as well as the proper scale value. The raw image data is stored pixel-by-pixel, line-by-line. Each pixel is an unsigned char (8-bit), a value of 0 means total black, a value of 255 means pure white. Each pixel is one of 256 shades of grey.

The normal structure of such a file is as follows:

Pos	**Count**	**Type**	**Description**
0	4	Integer	width of the video frame
4	4	Integer	height of the video frame
8	width*height	Array of Char	raw image data
width*height+8	8	Double	scale value

The structure repeats for the number of frames that are saved into an IMM file. All variables are saved in Little Endian Format. One has to convert the numbers when loading in an IMM file on a platform with a different Endian format (e.g. Sun).

An algorithm for loading in an IMM file is as follows:

- open the file
- determine file length
- read in the first four bytes and store as width
- read in the next four bytes and store as height
- calculate number of images inside the file
  (num = length / (width*height+16))
- seek to start of file
- repeat for every frame in the file
  - read in width and height
  - read in frame bits (width*height bytes)
  - read in scale
- close file

BKG File Format

The BKG file format is based on the IMM file format, except for two differences.

1. There is only one image inside the file.
2. There is no scale value saved.

All frames go uncompressed into the file. Together with the image data, there are some special variables saved into the file. These are width and height of the image. The raw background image data is stored pixel-by-pixel, line-by-line. Each pixel is an unsigned char (8-bit), a value of 0 means total black, a value of 255 means pure white. Each pixel is one of 256 shades of grey.

The normal structure of such a file is as follows:

Pos	**Count**	**Type**	**Description**
0	4	Integer	width of the video frame
4	4	Integer	height of the video frame
8	width*height	Array of Char	raw image data

The two variables are saved in Little Endian Format. One has to convert the numbers when loading in an BKG file on a platform with a different Endian format (e.g. Sun).

An algorithm for loading in an BKG file is as follows:
- open the file
- read in the first four bytes and store as width
- read in the next four bytes and store as height
- read in frame bits (width*height bytes)
- close file

4. Compile Guide

In this section everything is explained that must be known concerning the compilation of the "Videoclnt2_base" project. In order to properly compile the video client, the following software is needed:

- Microsoft Visual C++ 6.0 SP5
- Microsoft Processor Performance Pack for Visual C++ 6 (vcpp5.exe) installed into Visual C++
- TINE protocol installed on the PC
- TINE ActiveX control ACOP (acop.ocx) properly registered on the PC

Inside the project, profiles for an Intel Compiler optimised version exist. If one uses this compiler instead of the Microsoft Compiler built into Visual C++, it is possible to gain a significant speed improvement. One has to choose 'Release_Intel' (for Pentium III) or 'Relase_Intel_P4' (for Pentium IV) as the profile and enable the Intel compiler. After this, the project can be recompiled using the Intel compiler.

When the workspace file ('videoclnt.dsw') in Visual C++ is opened the first time, one should check whether the TINE ActiveX control "ACOP" is installed properly. One can check this by opening the resources and select the main dialog (Dialog->IDD_VIDEOCLNT_DIALOG). If no error message pops up that the ActiveX control could not be found, one can be pretty sure that it is installed properly. If an error message pops up maybe the control is just not registered properly. Normally it resides in
"<WINDOWS HOME DIR>\system32\". The filename is "acop.ocx". To register it one has to execute "regsvr32.exe acop.ocx" on the command line. If the control is not found or an error message pops up while registering, one should consider reinstalling the TINE protocol.

Once it is verified that the control is registered properly, the next step would be to choose the proper profile and rebuild the project (using Build->Rebuild All). Compiling the sources may take a while. If there was an error, one should check the requirements once again. Please note that there might be errors if the Intel profiles are compiled with the Microsoft compiler or vice versa.