LIVE 3D TV STREAMING

A Thesis
Presented to the School of Computing and Communications
of Blekinge Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Bachelors of Science
School of Engineering

by
Bishal Neupane
Pooya Moazzeni Bikani
June 2012

Supervisor: Dr. Siamak Khatibi
Examiner: Dr. Sven Johansson
ABSTRACT

The world is not flat as a pancake. It has height, width and depth. So we should see it even on TV. So far we cannot see three-dimensional programs directly into our TVs. Not even in cinemas with 3D cinema works "for real". For still there the magic sits in those glasses. The glasses of different colors allow distinguishing right and left eye impression tightened so that one sees different images with each eye. That is what creates the illusion of three dimensions. The goal of this thesis is to be on track to change that. Then you should achieve the same feeling without having required glasses, however, with a different technique. Do you remember those pictures that used to accompany the cereal packets? When angled in one direction, it was Donald Duck and angled it the other way it was Mickey Mouse. Our work is in the same way, though not with different images but with different perspectives. Same ribbed surface that existed at the pictures in cereal packets, are used as matter of fact on our 3D TV screen. Depending from which angle you look certain image information is hiding as it falls behind the ribbed surface. It thus separates views through the screen.

This thesis project is focused on a prototyping of live 3D TV streaming application where a live video of a scene is viewed on a 3D auto-stereoscopic display that gives two different perspectives, or views, simultaneously. The TV uses a face search (eye tracking) system to set up the television optimal for those who want to see 3D without glasses. During thesis a simple 3D studio was built where the focus has been to show depth perception. For scene capturing two cameras were used. We have found an engineering solution to take pictures simultaneously from the cameras. The input images from two cameras are sent to an analog to digital converter (frame grabber) as two channels of a virtual color camera, which means real time and synchronized capturing in a simple way.

The project has several applications written in C++ using various open source libraries, which essentially grab stereo image sequences from cameras using frame grabber, transfer image sequences to other applications via server communication, and display the live video in 3D display by exclusive rendering method. The communications between different applications for the purposes of transmission and receiving of video data is done using socket programming. The results of the project are very promising in which the live video of a scene can be viewed with noticeable depth despite obvious lagging in video timing.
ACKNOWLEDGMENTS

First and foremost, we would like to convey our utmost gratitude to our supervisor, Siamak Khatibi, for giving us an opportunity to work on this project. He certainly inspired and motivated us with his invaluable knowledge and experience; without whose diligence to help and teach, we would have not managed to complete the project in time. We would like to show our greatest appreciation to Bitra Shridhar. We cannot simply ignore the support and help we received from him in all times during the project. Finally, we take this opportunity to thank our families and friends for their love and support. Besides, we have seen everybody directly and indirectly involved in this project as contributors who genuinely made us feel proud.
# TABLE OF CONTENTS

Acknowledgements..................................................................................................................iii
List of Figures..........................................................................................................................vi
List of Tables............................................................................................................................vii
List of Abbreviations.................................................................................................................viii
1. Introduction............................................................................................................................1
2. Auto-Stereoscopic Display....................................................................................................3
3. Image Acquisition..................................................................................................................5
   3.1 Cameras...............................................................................................................................5
      3.1.1 The Camera and its Properties.....................................................................................5
   3.2 Frame Grabber.......................................................................................................................7
      3.2.1 Frame Grabber and Camera Connection......................................................................7
   3.3 Intellicam............................................................................................................................7
      3.3.1 Generating DCF Using Matrox Intellicam.................................................................10
   3.4 Matrox Imaging Library.....................................................................................................12
      3.4.1 Initialization..................................................................................................................12
      3.4.2 Acquire Images.............................................................................................................14
      3.4.3 Close Application.........................................................................................................15
4. Video Streaming.....................................................................................................................16
   4.1 What is Socket?..................................................................................................................16
   4.2 wxSocket and Other Socket APIs.....................................................................................17
      4.2.1 Socket Classes and Functions....................................................................................17
   4.3 Communication Protocol...................................................................................................18
      4.3.1 Server Application Starts Listening..........................................................................19
      4.3.2 Client 1.......................................................................................................................22
      4.3.3 Client 2.......................................................................................................................23
5. Rendering 3D Streaming Images...........................................................................................24
   5.1 Autostereoscopic Technique.............................................................................................24
   5.2 Background/Initialization of the SDK...............................................................................26
      5.2.1 How Does Rendering Work?.......................................................................................26
      5.2.2 Initialization..................................................................................................................26
   5.3 Input Image Data...............................................................................................................27
   5.4 Rendering..........................................................................................................................27
6. 3D Streaming Software in Action........................................................................................29
7. Tests and Results....................................................................................................................31
8. Conclusion..............................................................................................................................33
9. Future Work............................................................................................................................35
10. References.............................................................................................................................36
Appendix-I Application Codes..................................................................................................37
LIST OF FIGURES

Figure 1: Pictures taken using two cameras viewed on autostereoscopic display ........................................3
Figure 2: Cameras mounted on the rig ..............................................................................................................6
Figure 3: Frames (fields) of standard RS-170A video with electrical voltage levels ...............................8
Figure 4: Vertical blanking of standard RS-170A video .............................................................................8
Figure 5: Line timing .....................................................................................................................................9
Figure 6: Intellicam synchronization signal settings ..................................................................................10
Figure 7: Video signal settings ....................................................................................................................11
Figure 8: Video timing and pixel clock settings .......................................................................................12
Figure 9: Inheritance diagram for wxSocketBase ......................................................................................17
Figure 10: Communication protocol for the project ..................................................................................19
Figure 11: Left and right images viewed on autostereoscopic display .....................................................25
Figure 12: Lines on the normal screen vs seefront display .....................................................................26
Figure 13: Opening the session with the server .........................................................................................29
Figure 14: Enter the IPv4 address of the server .......................................................................................30
Figure 15: Start live transferring of the image data ..................................................................................30
Figure 16: An image of the scene created to perceive depth .....................................................................32
Figure 17: Live streaming of the scene .......................................................................................................32
LIST OF TABLES

Table 1: Time taken for “client1” application to grab and send images ........................................31
Table 2: Time taken for “client2” application to receive and render the images .................................31
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>Two Dimensional</td>
</tr>
<tr>
<td>3D TV</td>
<td>Three Dimensional Television</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CCD</td>
<td>Charged-coupled Device</td>
</tr>
<tr>
<td>CCIR</td>
<td>Committee Consultatif International Radiotelecommunique</td>
</tr>
<tr>
<td>DCF</td>
<td>Digitizer Configuration File</td>
</tr>
<tr>
<td>EIA</td>
<td>Electronics Industry Association</td>
</tr>
<tr>
<td>GmbH</td>
<td>Gesellschaft mit beschränkter Haftung</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>Hsync</td>
<td>Horizontal Synchronization Pulse</td>
</tr>
<tr>
<td>IDs</td>
<td>Identifications</td>
</tr>
<tr>
<td>IM</td>
<td>Instant Messaging</td>
</tr>
<tr>
<td>MIL</td>
<td>Matrox Imaging Library</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Open Source Computer Vision</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PCs</td>
<td>Personal Computers</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>VRPN</td>
<td>Virtual Reality Peripheral Network</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>Vsync</td>
<td>Vertical Synchronization Pulse</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

This thesis project is part of the degree program Bachelor of Science in Electrical Engineering at Blekinge Institute of Technology comprising total of 15 ECTS credits. The project was performed in a group of two students during the final year of studies. The project involves design and development of prototype software for grabbing images from two cameras at the same time; and streaming those image data over the network to a remote host for viewing using 3D display without requiring any headgears or glasses.

As recent as 2010, there have been unprecedented amount of interest and commercialization of 3D TV broadcasting. Like never before, it is now possible to get 3D experience in our living rooms without much of a hassle and fuss. With the advent of fine technologies, satellite cable service providers like DIRECTV have started broadcasting 3D channels. It has been further fuelled by the determination shown by many TV channels like, ESPN, SKY, Canal+, etc. to provide more and more 3D content to viewers. However, it still remains an area of further investigation and innovation because of the problems associated with troublesome glasses on the viewer’s side and lack of 3D content as of yet. Some people feel headaches and severe fatigue already when constantly exposed to 3D display with glasses. Thus, it comes to us as a great enthusiasm to be working on a project which avoids any use of external viewing aid on the viewer’s side – thereby making it auto 3D or formally known as autostereoscopic.

In this thesis project, our aim is to be able grab two images of the same scene from a slightly parallel translation in real time, thus making it live, and showing them together in 3D display without requiring any glasses. In the project setup, we have three applications running at the same time, namely two clients and a server. Firstly, one of the client applications grabs the stereo images from two cameras using a frame grabber. In addition, it handles the transfer of those image data to the server in real time. At the server end, the application in turn handles the transferring of data to another client which renders the images in 3D display device from SeeFront GmbH. In this way, we are trying to emulate a general live 3D TV streaming scenario.

Previously, similar type of project was performed using Canon EOS 450D cameras mounted on a rig. However, everything was handled on the same computer rather than using existing network infrastructures to send and receive data over the network. Besides, the size of the cameras was extravagantly larger and length of the USB cables shorter; so it is the main aim of this project to fulfill all those shortcomings by using smaller cameras, longer cables and transmission/receiving technologies to frame a live 3D TV broadcasting system.

The prototype software for the project is written solely on C++ while making heavy use of cross platform libraries, such as OpenCV, wxWidgets etc. The software is tested on PCs running Windows XP, 32-bit. However, should the need to migrate to another operating system occur, it should be fairly straightforward because of the use of cross-platform libraries in the project.

1
In the background section we discuss about the method of 3D display technique without the use of any glasses or headgears plus stereoscopic technology. Furthermore, a detailed description of how we acquired two left/right images for auto stereoscopic display is given in chapter 3, followed by a short discussion of transmission protocol used and 3D rendering of images in subsequent chapters. Moreover, in chapter 6, a brief insight is given into the behavior of the software when it’s running. Finally, we sum up the report with analysis of tests performed and results obtained in real time setup of the project scenario before the conclusion is drawn.
Chapter 2

AUTO-Stereoscopic Display

Auto-stereoscopic display technology omits the usage of 3D glasses to view 3D video scenes or images. As a matter of fact, auto-stereoscopic display devices can use additional lenticular lenses adjusted on the surface of the screen to project the content of display to observer to view different images with different eyes. Such a method of viewing technique is formerly known as binocular vision, i.e. depending on two eyes to perceive depth cues like it happens in normal human visual system. As we recall, stereoscopic technology depends on two images of the same scene taken from a slightly displacement by parallel translation; in turn 3D display devices use techniques such as, lenticular lens, parallax barrier etc. to separate left image to the left eye only and right image to the right eye only from the same screen [1].

Basically, what we need is just two pictures of the same scene from parallel translation to display it in 3D using one of many 3D display devices available in the market. Also, there are different types of single-viewer supported devices available for such a purpose. But, here in the project we only work with a single-viewer 3D display device from SeeFront GmbH as it uses lenticular lenses and special rendering engine to produce the stereo effect.

There are several methods by which one can get two pictures needed for 3D effect using auto-stereoscopic technology. For example, using a single camera and shift it parallel on the either side to get two pictures or using two cameras instead of just one, as the former method only works well for non-moving objects or scene. Thus, we are using two cameras to acquire images from a fixed position while keeping the distance between two cameras, called as base line, as small as the average of the distance between human eyes.

![Diagram](Figure 1: Pictures taken using two cameras viewed on autostereoscopic display)
As can be seen from the figure 1, there are three objects on the scene. More importantly, the 3D screen is where the focus point of the viewer is, which is also referred to as the zero-plane. The display devices use one of the earlier mentioned techniques to make objects appear, behind the screen or in front of the screen or just about on the screen. The distance between two corresponding points, points in the images which are points of the same object in 3D scene, is called parallax. So, as we see from the figure, object 1 is seen in front of the screen, because of negative parallax caused by intersection of the optical rays in front of the screen, i.e. also a viewer’s space. Likewise, object 3 is seen behind the screen, due to positive parallax, caused when the optical rays are intersected behind the screen. Last but not least, zero parallax occurs when the rays intersect somewhere on the screen causing the object to appear just on the screen [2].
Chapter 3

IMAGE ACQUISITION

In this chapter, we are going to describe the methods and tools used to acquire the left and right images of the scene required to perceive depth using binocular vision. We have used two CCD cameras for capturing the live video of the scene from a slightly different parallel translation. The type of cameras and its properties follow in the coming section of this chapter.

Moreover, a frame grabber is required to digitize the images from our analog cameras used in the project. We have used Matrox MeteorII/CH4 frame grabber for such purposes. There is a discussion about the use of the frame grabber and its various functionalities in section 3.2. Furthermore, we have shown how a DCF file can be used to initialize the key properties for the frame grabber before it’s used in the application. For the purpose of creating and configuring a DCF file, an interactive tool called Matrox InteliCam is used. The application is included in the MIL toolkit, a software product of Matrox Company.

3.1 CAMERAS

In this section, we explain the properties of the cameras used in the project. As we wanted to grab and display stereoscopic 3D video, we required two cameras. We have used two cameras but just one frame grabber to acquire left and right images at the same time. A brief explanation of how and why this was done is given in section 3.2.1.

One of requirement of cameras was that they should be small enough, so they could be placed fairly close together. This means with less base line we could have wider overlapping field of view. Also with less base line we are approaching to mimic the distance between eyes. Another requirement was that there should be limited differences between two cameras. Here mostly the bottle neck is the camera lenses. We had relatively the same quality for our camera lenses.

3.1.1 The Camera and its Properties

We have used CS 8600 series Toshiba Teli CCD (Charge-Coupled Device) cameras. The exact model is CS8620HCi. In this subsection, we give more insight into the properties of these family cameras and how they can be used to create a DCF file. A DCF initializes the video properties for the digitizer to make it enable grabbing frames and we will need them later in this project [4].

**TV System:** The camera supports both **EIA** and **CCIR** standards.

**EIA** stands for Electronics Industry Association, this association provided the standard for black and white television in the USA, Canada and Japan. This standard is referred to as RS-170.

**CCIR** stands for Committee Consultatif International Radiotélécommunique. This is the committee which provided the standards for black and white television used by most of Europe, Australia and others.
We call the devices and equipment which are compatible with black and white standards CCIR compatible [5].

**Image Sensor** of the camera is interline CCD.

The camera is using CCD technology for imaging. In a CCD image sensor, the coming light through the lens of the camera has different energy level (different number of photons). The light hits the, CCD array which gathers the energy of photons and converts it into electrical signals or voltage variation. By this way each CCD element represents a pixel where its intensity is the amount of converted voltage from photon energy. This technology is capable of capturing black and white images. Our cameras are black and white cameras which they use CCD technology. [6]

*Scanning Lines: 525(EIR) / 625(CCIR)*

*Scanning Format: 2:1 interlaced.* A thorough description of interlaced scanning will be given in section 3 of this chapter.

**Resolution:**

<table>
<thead>
<tr>
<th></th>
<th>EIA</th>
<th>CCIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>570 TV lines</td>
<td>560 TV lines</td>
</tr>
<tr>
<td>V</td>
<td>485 lines</td>
<td>575 lines</td>
</tr>
<tr>
<td></td>
<td>(TV lines)</td>
<td>(TV lines)</td>
</tr>
<tr>
<td></td>
<td>(V)</td>
<td>(V)</td>
</tr>
</tbody>
</table>

The complete list of properties of the camera is provided in Appendix as camera datasheet.

Now that we know some of the most important characteristics of the camera, after a short explanation about the frame grabber we give a complete instruction on how to create DCF file using these information of camera.

The cameras used in the project are shown here in figure 2.

![Figure 2: Cameras mounted on the rig](image-url)
3.2 FRAME GRABBER

What is a frame grabber? Frame grabber is a device used to capture (sample and quantize) video or images from analog or digital graphic components. In this project the frame grabber is used to acquire analog video stream from CCD interlaced scan cameras. Frame grabbers usually have the capability to capture, store or even process video streams and then store it on the memory of PC or transmit it in compressed format or as raw data.

Many cameras have the capability to be connected to PC via USB cable or via Ethernet. The camera which we were using in the project anyway doesn’t have this capability. So, we need to use frame grabber to acquire frames from the camera.

The frame grabber we are using here is Matrox MeteorII / CH4. It is a frame grabber for standard monochrome or color video acquisition.

3.2.1 Frame Grabber and Camera Connection

In our project, we required two image sequences, thus instead of just getting one color image from just one camera, we connected the cables between cameras and frame grabber in such a way that two monochrome images were grabbed as a color camera by the frame grabber at the same time; we made use of just two channels while completely avoiding the third one. In this way, we were able to capture two monochrome image sequences of the scene without any time latency between them which is crucial for 3D live streaming.

This way of connecting the cameras by just one cable using two different channels enables us to get two synchronized images. Instead if we connected the cameras by using two cables as the best we would be able to get two images with at least one frame delay and even this one is just possible through complicated image processing techniques inside frame grabber. So this connection was a tricky “engineering” part of the project.

3.3 INTELLICAM

Matrox Intellicam is an interactive program for interfacing cameras and Matrox frame grabbers. It allows the user to do fast checking of all the functionalities of different cameras and frame grabbers.

It can also be used to make different digitizer configuration format files which we use it later to adjust the initial setting of our grabber in MIL codes.

The Matrox Intellicam software comes along with MIL-Lite which is the library we used in this project [7].

Different settings can be adjusted by means of the Intellicam like the synchronization signal, pixel clock, video timing, and different aspects of video signal which we are going to explain it in this section.
Here we explain some of the terminologies used for interfacing cameras and the way they can be adjusted by Matrox Intellicam.

(Here the signal RS-170 is used as sample to explain some properties of video signals)

**Blanking Intervals**

A video signal has vertical and horizontal blanking intervals. At blanking intervals the voltage falls down to zero voltage so the video will be blanked at those periods and data stream is interrupted.

![Blanking Interval](image)

*Figure 3: Frames (fields) of standard RS-170A video with electrical voltage levels*

**Vertical Blanking**

It is kind of blanking which happens between two consecutive frames of video signal. It consists of back porch and front porch and also the vertical synchronization (Vsync) portion of blanking interval.

![Vertical Blanking](image)

*Figure 4: Vertical blanking of standard RS-170A video*
**Horizontal Blanking**

This type of blanking occurs between two consecutive lines. It consists of the front porch of the previous line, the horizontal synchronization pulse (Hsync) and the back porch of the current line.

![Figure 5: Line timing](image)

**Sync Pulses**

As we explained earlier we have two types of blanking intervals. The vertical blanking contains Vsync pulse which separates two fields or frames and it shows the start of new frame.

The horizontal blanking interval contains Hsync pulse, which separates two consecutive lines and shows the start of the new scanning line [4].

**Pixel Clock**

Pixel clock is a timing signal which is used to divide each line into pixels. Some cameras provide pixel clock themselves, but it is also possible to use the frame grabber to generate the pixel clock as we desire.
3.3.1 Generate DCF by Matrox Intellicam

Maybe the most important task that can be done by Matrox Intellicam is generating DCF files which can be done very fast and be used in MIL codes to avoid long coding for adjusting the settings.

For synchronization signal a digital signal generated by frame grabber is used which is responsible for generating both Hsync and Vsync.

Now in the figures below you can see different settings we have used to create our DCF file.

Figure 6: Intellicam synchronization signal settings
The video signal settings are shown in the figure below. The original signal is analog and has three channels which we will use two of its channels.

Figure 7: Video signal settings
For video timing and pixel clock settings we found optimal results with the parameter values which are shown in the following figures in the next page.

![Figures showing video timing and pixel clock settings](image)

Figure 8: Video timing and pixel clock settings

### 3.4 MATROX IMAGING LIBRARY

In order to be able to use a frame grabber to acquire images from the cameras, we used Matrox Imaging Library (MIL). MIL features a collection of software tools for developing machine vision, image analysis and medical imaging software applications. It also features interactive software and programming functions for image capturing, processing, analysis, annotation, display and archiving [3].

After making suitable configuration by creating right DCF file for our need, we used the MIL and OpenCV to grab the images from the cameras by the frame grabber. The MIL has especially functions which are designed to grab images from grabber and achieve processing on acquired images. The following is the explanation of how our program does to grab two monochrome images from the two CCD cameras by implementing the two channels of one color video stream.

#### 3.4.1 Initialization

First we need to prepare the frame grabber for grabbing mode when we are intending to capture a video. Some settings can be set in the DCF file and is usually the safer and easier option; however, it is also possible to prepare the frame grabber entirely by using functions and parameters in the C++ application itself.
In our application, we have used a DCF file for most of the configurations of the frame grabber. We configured and tested the DCF file using Matrox Intellicam - an interactive tool from MIL. Let’s briefly look at the general workflow of the application at its simplest form.

Firstly, we have to initialize the overall application by allocating one through the use of `MappAlloc` method.

```c
MappAlloc (M_DEFAULT, &MilApplication);
```

Here, `MilApplication` is just an identifier to MIL application. We have allocated and started an application in its default form here.

In addition, we have to allocate the resources for the system we are using, i.e. the host computer. For that we need to use `MsyssAlloc` method with various parameters like, name of the frame grabber, frame grabber number and the system identifier.

```c
MsyssAlloc(M_SYSTEM_METEOR_II, M_DEF_SYSTEM_NUM, M_SETUP, &MilSystem);
```

This function starts the system including the computer we are working on and the frame grabber.

After the allocation of the system, we allocate the digitizer so that the application can use MIL functionalities. This is the place where the DCF file is used by the application to initialize the frame grabber for image acquiring usage and more. We pass the location and name of the file as an argument to the function `MdigAlloc`. In a nutshell, this method checks for the existence of the digitizer and starts it for further use.

```c
MdigAlloc (MilSystem, M_DEFAULT, DCF_NAME, M_DEFAULT, &MilDigitizer);
```

When we are done with the initialization of the application, system and the frame grabber, it is now time to allocate the buffer to store the picture. We use `MbufAllocColor` method to set apart a chunk of memory for storing purposes.

```c
MbufAllocColor (MilSystem, MdigInquire (MilDigitizer, M_SIZE_BAND, M_NULL),
                (long) (MdigInquire (MilDigitizer, M_SIZE_X, M_NULL)),
                (long) (MdigInquire (MilDigitizer, M_SIZE_Y, M_NULL)),
                8L+M_UNSIGNED,
                M_IMAGE+M_GRAB+M_DISP,
                &MilImageDisp);
```

Evidently, original images taken by the frame grabber are colorful and have three bands. This function allocates the color buffer on the system to store those frames.

Here the function `MdigInquire` is being used to extract the information about the number of bands and the size of the frames from grabber. These settings are available through the use of the DCF file earlier.
Moreover, this buffer is used to grab the image and display it, so we have M_GRAB+M_DISP parameters set as arguments.

We also setup 2D buffer, just in case, to extract each band of the three band color image. For this purpose, we make use of the method MbufAlloc2d as shown below.

\[
\text{MbufAlloc2d (MilSystem,}
\]
\[
\text{(long) (MdigInquire(MilDigitizer, M_SIZE_X, M_NULL))},
\]
\[
\text{(long) (MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL))},
\]
\[
8L+M_UNSIGNED,
\]
\[
M\_IMAGE+M\_GRAB+M\_DISP,
\]
\[
&MilTempbuf);
\]

This function operates in the same manner as the one before but the only difference is that it allocates a single channel buffer instead of three. It will be used later to buffer each band of the original color image.

\[
\text{MdigControl(MilDigitizer, M\_GRAB\_FIELD\_NUM , 2 );}
\]

For interlaced videos the parameter M\_GRAB\_FIELD\_NUM specifies the number of fields to be grabbed when used like shown above and passed as an argument to MdigControl method.

\[
\text{MdigControl(MilDigitizer, M\_GRAB\_START\_MODE , M\_FIELD\_START );}
\]

Here, in the above function, we set the grab start mode to any field by putting the value M\_FIELD\_START for M\_GRAB\_START\_MODE setting, which helped to get better synchronized images.

It is possible to free the resources by clearing the display buffer as shown below:

\[
\text{MbufClear(MilImageDisp, 0);}
\]

3.4.2 Acquire Images

When the MIL application, the frame grabber and buffers are properly initialized, we are ready to grab the images.

Before we grab any image; we can always clear the buffer so we do not see any unexpected results. So, we clear the buffers designated to store the color and monochrome image data by using the method MbufClear and passing the name of the buffer as an argument.

\[
\text{MdigChannel (MilDigitizer, Channel);}
\]
\[
\text{MdigGrab (MilDigitizer, MilImageDisp);}
\]

Now, as seen above, we use MdigChannel method to set the channel number from which to grab the image. Next, MdigGrab method grabs the image on the buffer MilImageDisp.
Moreover, to separate each band of the color image and get two 2D images (monochrome), we copy the red, green or any two, depending on our specific settings to single channel buffer using the method `MbufCopyColor`.

```
MbufCopyColor(MilImageDisp, MilTempbuf, M_RED);
MbufGet(MilTempbuf, left_image->imageData);
```

As it is seen from above piece of code, we copy each band of the color image in the temporary buffer and from there get the data and store into an `IplImage` object, i.e. `left_image`. Similarly, we copy the green band to another `IplImage` and use them to send to the server.

### 3.4.3 Close Application

At the end, to maintain the application stability and not run into memory resources related problems we have to free the memory, system, frame grabber and close the application by using these simple commands.

```
MbufFree (MilImageDisp);
MbufFree (MilTempbuf);
MdigFree (MilDigitizer);
MsystFree (MilSystem);
MappFree (MilApplication);
```
Chapter 4

VIDEO STREAMING

In this project we used a typical protocol to successfully transfer 3D video stream (image sequences) from a dedicated client to a server which in turn handles the transferring of such stream to other clients. However for 3D video streaming purpose there were different options which could fulfill the requirements, either using high gain wireless antennas to make up point to point connection or using the popular internet protocol, TCP/IP. Due to time constraints, we used internet and its underlying protocol TCP/IP, to fulfill our requirements on this project. For the purpose of creating GUIs for our applications we have used wxWidgets, a widget toolkit, which is cross-platform, free and open source. Moreover, it is written in C++ and is being actively developed since its first release back in 1992. In the first section, we are going to explain about sockets and socket programming to achieve fast data transfer using different protocols. Additionally, we will discuss about wxWidgets wrapper classes and how to use them to establish a communication between clients and server in the following sections. For convenience, we used wxWidgets’ wxSocket API for socket programming in the project because its implementation is platform-independent and we can quickly move between operating systems.

4.1 WHAT IS SOCKET?

Whenever we talk about computer networking a socket is regarded as the most fundamental technology. Communication between applications using sockets is possible through the use of standard execution built into network hardware and OS. Despite network software technology being dubbed as red-hot “Web” phenomenon, the use of socket in networking has been there for a long time now, more or less two decades [8].

As the popularity of internet has kept growing ecstatically, so has the need to communicate between software applications on the network, like when we use the web browsers to access web pages or while chatting via IM or when sharing files on a peer to peer communication systems. One important property of sockets is that it does not matter whether you want to send images or text files or any other data, it simply ignores the format and the content type. Moreover, the communication does not necessarily have to be established between computers, it can be any pair of devices supporting sockets.

The socket API was originally a part of the Berkeley Software Distribution Unix operating system, and because that socket API originated from only one source, it has become the standard. All modern operating systems offer a socket layer, providing the ability to send data over a network (such as the Internet) using common protocols such as TCP or UDP. Using wxWidgets’ wxSocket classes, you can reliably communicate any amount of data from one computer to another [9].
4.2 WXSOCKET AND OTHER SOCKET APIs

Even though the basic socket features and functions are very similar on Windows, Linux, and Mac OS X, each socket API implementation has its own nuances, usually necessitating platform-specific tweaks. More importantly, event-based sockets have very different APIs from one platform to the next, often making it a significant challenge to use them. The wxWidgets provides socket classes that make it easy to use sockets in advanced applications without having to worry about platform-specific implementations or quirks [9].

As we are heavily using wxWidgets in our project like as making simple GUI or using OpenGL extension to generate textures to feed to the renderer, we decided to use wxSocket instead of native API for Microsoft operating systems i.e., Windows Sockets (winsock). This choice has the advantage of that we do not need to worry about platform specific implementation as wxSocket hides bulky initialization codes from the program interface.

4.2.1 Socket Classes and Functions

As we can see from the figure 9, wxSocketBase lies at the core of socket operations, which enables the application to establish a connection and send or receive data, check for errors, terminate the connection and so on. Although, old wxWidgets version 2.8.9 does not support datagram sockets, the newer version of the framework does give full support to datagram socket operations based on UDP/IP protocol.

Everything in wxWidgets is handled through events and socket operations are no exception to that, avoiding the necessity to have a separate thread for socket operations. Typically, it has wxSocketServer and wxSocketClient to provide listening server and connecting client respectively. And finally, wxSocketEvent is used to notify the application of an event that has occurred on a socket [9]. We can either use sockets from the main GUI thread and subscribe to
events using `wxSocketEvent` handler and avoid the need to use threads or we can use sockets in a separate threads and block the GUI while using it. We can raise or disable so called “Socket Flags” parameters and totally change how the socket behaves. A brief discussion of “Socket Flags” will follow in a later section.

### 4.3 COMMUNICATION PROTOCOL

Here we discuss in detail about how the socket programming is done using wxWidgets’ `wxSocket` API for our purpose of sending 3D video stream from one client to a server and from server to a remote host.

Firstly, we had prototyped two applications in the project to fulfill the requirements. To avoid ambiguity with client and server notation, we are simply going to refer to our applications as “client 1” and “client 2” because in a very basic setup a client requests connection to a server whereas server is listening on its port for possible clients. As soon as the connection is established, data can be transferred either way, so the concept of client-server connection does not make sense unless we are talking about alternative design which we will explain about it shortly afterwards.

In this client-client setup, the application named “client 1” was using MIL library to grab images from cameras and send it to client 2 over TCP/IP socket connection. At the “client 2” end of application, as soon as images are received, there is a possibility that renderer will use those images to show them in 3D display. However, this approach led to rather unpleasing taste of design where both of these applications had to be connected to external hardware, for example a frame grabber to grab images or 3D display to render the images. Moreover, the problem was that unless the computer running “client 1” application has a real IP (public IP address) we would not be able to communicate with a remote client and both applications had to be running on the same machine which did not sound as a design for live streaming of 3D video. Also another fact that we did not have public IP on the system which we grabbed images meant we needed a server at remote location with a public IP address if we were to be able to communicate with remote clients.

So, implementing the classical client-server approach, we just needed to have one “client 1” application and one server application which in turn managed streaming of data to other clients with 3D display.
4.3.1 Server Application Starts Listening

Okay, now according to our design we have a server application which waits for connection from clients. Let’s look closely at some of the useful functions and classes used in the server application to instantiate a socket for communication with clients. To keep it simple we only talk about the interface for socket programming here.

As seen previously, if our application needs to listen for incoming client connection requests we have to declare an instance of `wxSocketServer` class which is derived from `wxSocketBase` class like below; notice we declared a pointer instance of the class for performance reasons.

```cpp
wxSocketServer *m_server;
```

Before we can start listening on this server socket, we have to assign IP address and the port number to listen to, so we may want to create an instance of `wxIPv4address` for that reason like this:

```cpp
wxIPv4address addr;
addr.Service(3000);
addr.Hostname(127.0.0.1);
```

By default the address is localhost, i.e., IP address of 127.0.0.1 or we can easily change the address by using `Hostname` method and passing IP address as argument like seen above. The method `Hostname` returns true on success and false if something goes wrong (invalid hostname or invalid IP address). Likewise, `Service` method helps us to set the port number to associate the socket with. This method also returns true on success and false if something goes wrong [10]. Usually it is wise to use a number more than 1024 as a port number because numbers between 0-1024 are reserved for services.
And now after configuring the address instance we are ready to create an instance of wxSocketServer by passing wxIPv4Address object on the fly,

```
m_server = new wxSocketServer(addr);
```

after creating an instance to wxSocketServer, we can use Ok() method to make sure server is ok and listening to client requests. And the important part comes, where we setup the event handler and subscribe to connection events first. The small chunk of code used for this in the program is shown below.

```
m_server->SetEventHandler(*this, SERVER_ID);
m_server->SetNotify(wxSOCKET_CONNECTION_FLAG);
m_server->Notify(true);
```

Before talking about how event handling works for sockets lets take a look at event table associated with our main frame (parent widget) in the server application. There are many different types of events occurring in a typical wxWidgets GUI application. When we talk about events associated with a particular socket, they are all filtered through EVT_SOCKET event handler.

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(SERVER_QUIT, MyFrame::OnQuit)
    EVT_MENU(SERVER_ABOUT, MyFrame::OnAbout)
    EVT_SOCKET(SERVER_ID, MyFrame::OnServerEvent)
    EVT_SOCKET(SOCKET_ID, MyFrame::OnSocketEvent)
END_EVENT_TABLE()
```

It can be seen from the example above that it has general format of EVT_SOCKET(IDENTIFIER, FUNCTION). Basically, it means that whenever an event occurs on the socket, the application uses the IDENTIFIER to guide events to the corresponding FUNCTION after we have set the event handler. Besides, the FUNCTION must accept wxSocketEvent as a parameter.

Using the methods SetNotify and Notify on the socket we can subscribe to only the types of events we want to process. Like seen above, firstly we just subscribe to connection events using, m_server->SetNotify(wxSOCKET_CONNECTION_FLAG). After this any connection requests from clients will be directed to the function associated with SERVER_ID, i.e., MyFrame::OnServerEvent [11].

Now, once the client application tries to connect to the server on the specified port using the valid IP address, it will trigger wxSOCKET_CONNECTION event and it is up to the server side to establish the connection by accepting it. See below:

```
wxSocketBase *sock;
sock = m_server->Accept(false);
```

Unsurprisingly, Accept method let’s server to initialize wxSocketBase object which represents server-side of the connection. More importantly, the argument for method Accept is passed as false which represents boolean value wait. This simply checks for any pending connection and returns immediately without blocking GUI. But once the application comes this far, there must be at least a pending connection because we just subscribed to wxSOCKET_CONNECTION earlier. If wait
parameter was \texttt{true} it would wait for client request for connection to arrive and also block the GUI. Thus, care must be made while doing so as GUI will behave unpleasently.

The return value from \texttt{Accept} method would either be opened socket connection or NULL if error occurred and connection could not establish [12].

As with server socket, it is easier to handle events on newly created socket connection with particular client and we can use the same event handler for many clients too. And the process is the same, except that we need a different unique identifier and function to route the event to. Again, the \texttt{FUNCTION} must accept \texttt{wxSocketEvent} as a parameter.

\begin{verbatim}
sock->SetEventHandler(*this, SOCKET_ID);
sock->SetNotify(wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);
sock->Notify(true);
\end{verbatim}

As shown above, we can subscribe to events from different clients using the same event handler, which is very convenient. Notice that using \texttt{SetNotify} method we are subscribing to \texttt{wxSOCKET_INPUT} and \texttt{wxSOCKET_LOST} events, these events occur when there is some data to be read on the socket and when the connection between client and server is broken down respectively. Moreover, we make sure to receive events by invoking \texttt{Notify()} method passing \texttt{true} as argument. In this way it is easier to subscribe or unsubscribe to events on a particular server-client socket without having to worry about event flags raised.

At this point, server has listened to client connections and accepted connections, subscribed to input and connection lost events, and we assume in our design that the first client which requests connection is actually going to send 3D streaming images. Firstly, input events and connection lost events get directed to,

\begin{verbatim}
void MyFrame::OnSocketEvent(wxSocketEvent& event);
\end{verbatim}

where, this method checks for unique test code by reading a single byte. And if the code is valid, it will call the following method but before we read anything we unsubscribe to input events using \texttt{SetNotify} and passing just the \texttt{wxSOCKET_LOST_FLAG} because we do not want to keep receiving input events in the middle of reading data,

\begin{verbatim}
void MyFrame::OnStreamVideo(wxSocketBase* sock);
\end{verbatim}

Here is the main part of the application where we actually read and guide data back to other connected clients (if there are any). First, we read the height and width of the incoming images, and create a new IplImage header so, we can find out the exact size of the incoming data. After we know the size of the pictures, the server reads image data from “client1” and sends it to “client2” through simple use of \texttt{Read} and \texttt{Write} commands.

Here, important thing to consider is the use of “Socket Flags” in proper way to facilitate what each application needs. We use \texttt{SetFlags(wxSocketFlags flags)} method on a given socket to customize IO operation. The \texttt{flags} can be a set of flags parameter ORed together.

The four main flags most commonly used are:
wxSOCKET_NONE - normal functionality
wxSOCKET_NOWAIT - read/write as much data as possible and return immediately
wxSOCKET_WAITALL - waits for all the data to be read or written unless an error occurs
wxSOCKET_BLOCK - it block the GUI part when performing read/write operations

You can refer to the wxWidgets wiki page for more detailed description on each of these flag parameters [13].

In the next section the topic of discussion will be clients and how they communicate with the server and for what purposes.

### 4.3.2 Client 1

The “client 1” is responsible of grabbing the images from the CCD cameras using Meteor-II/MC4 frame grabbers and sending it in a regular frame rate as needed, typically less than 25fps.

As with the server we have to declare an instance of `wxSocketClient` class which is derived from `wxSocketBase` class and thus inherits all of its functionalities. We declare and create it on the fly like,

```cpp
wxSocketClient *m_sock;
m_sock = new wxSocketClient();
```

After creating the client socket we can use event handler as with server socket and subscribe to convenient events, refer to earlier section on how to achieve this. On the other hand, after the successful creation of the socket we would like to connect to a server assuming server has started listening on its port. To connect to a server we need to have an instance to `wxIPv4address` class that which we can pass as an argument to `Connect` method on the socket after configuring valid IP address of the server and a valid port number on which sever is listening to, see below.

```cpp
wxIPv4address addr;
addr.Hostname(IPv4 address);
addr.Service(3000);
m_sock->Connect(addr, false);
m_sock->WaitOnConnect(10);
```

There are actually two ways in which a client can request a connection from a server. One way around this is to call `Connect` and `WaitOnConnect` like exactly shown above. Here the second argument on `Connect` method means whether we should block GUI when waiting to establish connection with the server or not. If the argument is `true` it waits until a connection gets established or we receive an error while blocking the GUI calls. But if the second argument is `false` we can simply follow with a timeout in seconds as an argument to the method `WaitOnConnect` which will just wait for the timeout to finish and result in unsuccessful connection or return immediately with successful connection.
In addition, after “client 1” is connected to the server, it is ready to stream the videos by periodically grabbing images and sending it using \texttt{Write} command on the socket. More details on how to grab the images using frame grabber are given on chapter 1.

\textbf{4.3.3 Client 2}

This end of the application is very similar to “client 1” except that it does not grab and send images instead receives the image sequence from the server and renders the images by 3D display engine. So, basically it follows the setup of “client 1” and requests connection from the server. Once the connection is established the client reads data from the socket buffer and renders it on 3D display as soon as possible.
Chapter 5

RENDERING 3D STREAMING IMAGES

In this chapter the focus of discussion will be on 3D display engine and implementation of its SDK to render left and right images using OpenGL extension of wxWidgets. What we had in the lab was a 19” SeeFront auto-stereoscopic 3D display driven by its special engine [14] to render 3D images. Basically, using SeeFront SDK we can integrate SeeFront functionality in any OpenGL / DirectX applications. Moreover, it has a very simple interface which is easy to use without much hassle. We are going to talk about some of the important steps and methods used in the application to successfully render two images in this chapter.

Currently the rendering of interlaced images on the 3D desktop display is done from our earlier mentioned “client 2” application. In this application, because of threading issues, we used wxTimer to achieve more appropriately without having to worry about application stability. At its core, the application is receiving the video stream (image sequences) from the server and making it available to the renderer. The details on how the rendering is done using SeeFront SDK is to follow in the coming sections.

Before going into the details of the programming interface it is recommended to read through following background section for more introspect on how the auto-stereoscopic display makes it possible to get depth cues without using external glasses or headgears.

5.1 AUTOSTEREOSCOPIC TECHNIQUE

As formerly discussed, auto-stereoscopic display is automatic in a sense that no additional devices such as glasses are required in order to view 3D content. The use of additional optical elements just on the surface of the screen attributes to the viewing of different images from each eye. To date, it is of great interest for the researchers to investigate and design the optimal and convenient methods of viewing 3D stereo images without headgears or glasses, i.e. auto-stereoscopic approaches. Here, we only discuss the lenticular approach to autostereoscopic display technique as our display device is based on such mechanism.

Lenticular lenses are cylindrically curved optics which allows magnification of images in such a way that different images are seen from slightly different parallel displacement at the same time. A sheet of such optical component is fitted on the display to manifest 3D viewing of stereo image pair. The sheet itself is transparent and the lenticules form an array of tiny glass or polyester lenses. The idea behind such technique is illustrated in figure 12 [1].
As seen from the figure 11, the left and right eyes see the left and right images respectively. In turn, two images are combined in brain to form one 3D image. There would be a huge drawback of using such a technique if the viewer is moving in position constantly and the display is not adjusted. For that reason, SeeFront display screen adjusts the display depending on the viewer’s eye position. The SeeFront rendering application is constantly communicating with the eye tracking software through VRPN protocol and changing the images accordingly when viewed from different angle in real time.

It is shown in figure 12 that the lines on the normal screen are not really parallel to the lines on the SeeFront’s display. However, as shown in the figure the lines on the normal screen make an angle of 18 degrees to the lines on the screen that aids viewing 3D autostereoscopic images and videos. In the figure all red lines represent the lines from SeeFront’s display which are all parallel to each other. In the same way, the blue lines represent lines from the normal display screen and they are all parallel to each other as well. The figure is not drawn to scale and is just showing a zoomed chunk of the screen.
5.2 BACKGROUND / INITIALIZATION OF THE SDK

In this section we start with a brief discussion of how rendering process is carried out in the application and the things necessary to start using SeeFront methods in our application. However, before we can start using SeeFront methods we must ensure that a SeeFront object is created. We will see in next sections how this is exactly done.

5.2.1 How does rendering work?

For the optimal image display the SeeFront application needs current eye position of the viewer using an eye tracking application which runs in the background. The eye tracking application is responsible for delivering of the eye position of the user constantly to the application. For the purpose of communication between an eye tracking application and the SeeFront application, VRPN protocol is used [15].

Moreover, during the rendering of the images one requires a rectangle with the size of the rendering window. Then, the rectangle is to be filled with the interlaced images; considering left / right images; eye position; position of the render window etc. At the time of rendering, a shader goes through each pixel in the rectangle and evaluates its correct position and intensity value. Refer to the SeeFront SDK for detail. However, one should be aware of the fact that SeeFront technique works well if it’s run on native resolution, i.e. 1600x1200 as in case of our display.

5.2.2 Initialization

In this subsection we discuss how SeeFront SDK can be used in OpenGL application. However, the interface is mainly similar between OpenGL and DirectX applications. Firstly, we have to create an instance of the SeeFront object before any other methods can be called upon it.

```cpp
sfogl::IlaceHandle sofgl::createInstance(void);
```
Upon calling this method, it returns SeeFront object. To avoid any memory leakage, or before creating another instance of the SeeFront object, a call to `sfogl::destroy()` method must be made. It is required that a valid OpenGL context exists before calling this method anyway. Refer to the SeeFront SDK for detailed description on other methods to create an instance of the SeeFront object [16].

After the successful initialization of the SeeFront instance, tracker communication between an eye tracking application and the application must be established.

```cpp
// set tracker updates to the interlacer
sfogl::setTrackerCallback(m_interlacer, &g_tracker_update);
sfogl::startTrackerUpdate(m_interlacer);
```

Here, `m_interlacer` is an instance of the SeeFront object, and `g_tracker_update` is the tracker callback. Apparently, these methods should be called in the order as seen above. More detailed description of the functions and their parameters as well as VRPN protocol are given on the SeeFront SDK documentation.

### 5.3 INPUT IMAGE DATA

Obviously, the main input data to feed are left and right images. Passing the image data is as simple as calling `sfogl::setTextures()`.

```cpp
sfogl::setTextures(m_interlacer, m_Images, 2, 0.0f, 1.0f, 1.0f, 0.0f);
```

Here, `m_interlacer` is the SeeFront object and `m_Images` is the images in the form of OpenGL textures. In an OpenGL application, two valid textures must be bind to some texture IDs using `glBindTexture()` after calling `glGenTextures()`. Next, the texture IDs that are bound are sent as an argument which has 2 entries. Accordingly, the first entry should be left image and the second entry should be the right image.

### 5.4 RENDERING

The ultimate step in the program workflow is rendering.

```cpp
sfogl::setTextureSize(m_interlacer, 1600,1200);
sfogl::setScreen(m_interlacer,0,0,1600,1200);
sfogl::render(m_interlacer);
SwapBuffers();
```

After setting the textures, we set the texture size in order to get the actual image ratio. Finally, the SeeFront SDK needs to know the resolution and coordinates of the SeeFront monitor. By calling `sfogl::setScreen()` method we can set these values.

Ultimately, the latest available tracker data and input parameters facilitate interlacing of the images and the frame buffer is filled with the appearing of new pixel position and intensity values. This is
always the last call using the method `sfogl::render()`. One final step before rendering is complete is to flip the memory page using `SwapBuffers()`.
Chapter 6

3D STREAMING SOFTWARE IN ACTION

The aim of this chapter is to give insight into how the overall application functions when it is run. We have mostly covered the individual functionalities of the applications earlier chapters. Here, we just try to relate the overall flow of the program with its performance.

The first and foremost thing is we need to have a dedicated server with a real IP in a remote location and two client applications on any computers connected to the Internet. Or, a simple way around to solve the problem of real IP would be to have all the applications on the same computer, not necessarily having a real IP. Either way the overall functionalities are the same, maybe except the rate at which images are transferred from one computer to another.

As discussed previously, there are three primary applications running at the same time when 3D images are being streamed and shown on the display.

The first application which needs to be up and running is the server. After the server application is up and running, it waits and listens for any potential client connections. As soon as the server is ready, we can run the “client1” application at a remote host and input the IPv4 address of the server in a dialog box that prompts.

![Image](image1.png)

Figure 13: Opening the session with the server

After that, if the server was ready, a successful connection is established and we see a confirmation. Similarly, we can run the “client2” application and establish a connection with the server. In our project, the first client which connects to the server is supposed to be the one which streams image data acquired from the cameras. So, to maintain the flow of the programs and not run into unexpected errors, “client1” must be connected to the server before “client2”.
As soon as both the clients are connected to the server, we are ready to start transferring of image data; which makes it possible for “client2” end of the application to render images in 3D display.
Chapter 7

TESTS AND RESULTS

Here, we have compiled an analysis on the current tests and results for the purpose of preview for someone who is interested in the relative work. Basically, we just want to show the time taken, in appropriate units, during grabbing, sending, receiving, and rendering of the images in live scenario. During the test, our “client1” and “server” applications were run on the same computer from Image Processing Lab and the other application: “client2”, which receives the stereo image sequences and renders in 3D display was ran in another lab next door. Upon establishing the communication between different applications, image sequences were transferred at a fairly decent rate (between 15-20 frames/s) and the viewing of the 3D video was impressive with good impression of depth. However, there was a noticeable lag in the streaming of the live video. But, overall the streaming worked better than anticipated.

In table 1 below, the average time taken by “client1” application to query images from the cameras and the average time taken to send the image data to socket is shown; both measures are in milliseconds. The numbers in the table below represent an average time taken to grab or send images for each 60 seconds interval. So, from the 1st column of table 1, it can be interpreted that it took (in average during 60 seconds) 62ms and 219ms to grab and send one set of right and left images respectively.

<table>
<thead>
<tr>
<th>Grabbing(ms)</th>
<th>62</th>
<th>63</th>
<th>78</th>
<th>79</th>
<th>63</th>
<th>62</th>
<th>78</th>
<th>93</th>
<th>78</th>
<th>77</th>
<th>78</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending(ms)</td>
<td>219</td>
<td>218</td>
<td>250</td>
<td>225</td>
<td>230</td>
<td>219</td>
<td>250</td>
<td>218</td>
<td>217</td>
<td>217</td>
<td>223</td>
<td>218</td>
</tr>
</tbody>
</table>

Moreover, average time taken during 60 seconds interval for reading and rendering of each image sequences by “client2” application is shown in table 2.

<table>
<thead>
<tr>
<th>Reading(ms)</th>
<th>78</th>
<th>79</th>
<th>79</th>
<th>79</th>
<th>78</th>
<th>78</th>
<th>79</th>
<th>78</th>
<th>79</th>
<th>78</th>
<th>78</th>
<th>78</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering(ms)</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>
Moreover, the figures of the scene and display engine are shown below when the application was fully functional. In figure 16 below notice the tracking of the viewer’s eyes to adjust 3D viewing.

Figure 16: An image of the scene created to perceive depth

Figure 17: Live streaming of the scene
Chapter 8

CONCLUSION

The main goal of the project was finally accomplished as we were successfully able to acquire the images, send them over the network, and render them on the 3D display. When the applications were ran and tested we got somewhat desired results. Initially we had the difficulty of synchronizing the two left and right images in parallax translation while using MIL. But with the help of sophisticated configuration of DCF we finally achieved perfect synchronization between images. After that it was all about streaming of the video data, which was done at a rate of about 15-20 frames/sec depending on the Internet speed, as it always varies. However, upon receiving the data and feeding it into the 3D engine, we could feel the 3D effect as well as the live streaming, even though initially there was noticeable lag in timing of the video frames.

The main shortcoming of the project was that the video streaming was black-and-white instead of color. The first reason behind such a design was related to insufficient and varying network speed while streaming video data. The other reason was the possible complexity associated with getting two synchronized image sequence by using completely different channels given the timing constraints for the project. Anyway, the overwhelming success in the current project set-up should give enough impetus for improvements by making use of reliable and faster point-to-point wireless network connection with antennas, as well as color video streaming instead of black-and-white.
Chapter 9

FUTURE WORK

This project was done by manipulating black-and-white images. Also, to transmit the acquired images the internet was used. Moreover, the resolution of the images was not that high. This project can be continued to be able to grab and send colorful images with higher resolution and instead of the Internet, wireless point-to-point communication mechanisms can be used.

The frame grabber which was used to grab images is also capable of grabbing colorful images. In addition, it is also possible to grab high resolution images with the device. So, it is not much more work to adjust the device to grab colorful images with high resolution which is of course better for human perception.

In this project frames are transmitted at the rate of about 15-20 frames/second, depending on the available Internet speed. We know that if we manage to get more rate for sending the frames it would be much better for humans to perceive depth in videos. Moreover, to facilitate the need to achieve higher transmission rate we can use different compression and decompression techniques.

Also, instead of using Internet to transmit the frames we can try to use some wireless communication protocols to transmit the frames without using cables. This would be more interesting for commercial usage because it makes it possible for users to watch the 3D streaming at any point without having any specific device for connection. Also, by using wireless communication it is possible to broadcast the stream and make many users able to watch the 3D stream at any point in a small area.

So many new improvisations can be done on this project to make it more useful for commercial usage. Providing colorful images with high resolution and using some compression schemes before transmission are some of those. Ultimately, using antennas with some point-to-point wireless transmission protocols makes it easier for users to enjoy the stream reliably.
Chapter 10

REFERENCES

[16] ‘The SeeFront 3D Software Development Kit v1.3’.
APPENDIX-I APPLICATION CODES

Client 1 Application

/**
\file parent_main.h
\author Bishal Neupane
\author Pooya Moazzeni Bikani
\author 2012 Blekinge Institute of Technology. All rights Reserved

(20 July 2012) Bishal and Pooya - Created file.
*/

// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef __PARENT_MAIN_H__
#define __PARENT_MAIN_H__

// -----------------------------------------------
// headers
// -----------------------------------------------
// For compilers that support precompilation, includes "wx/wx.h".
#include "wx/wxprec.h"
#if defined (__BORLANDC__)
  #pragma hdrstop
#endif
// for all others, include the necessary headers
#ifndef WX_PRECOMP
  #include "wx/wx.h"
#endif
#include "wx/socket.h"
#include "wx/wfstream.h"

// -----------------------------------------------
// resources
// -----------------------------------------------
// the application icon
#if defined(__WXGTK__) || defined(__WX11__) || defined(__WXMOtif__) || defined(__WXMAC__)
  #include "mondrian.xpm"
#endif

// Define a new application type
class ClientParentApp : public wxApp
{
public:
  virtual bool OnInit();
};
#endif __PARENT_MAIN_H__

/**
\file parent_main.cpp
\author Bishal Neupane
\author Pooya Moazzeni Bikani
\author 2012 Blekinge Institute of Technology. All rights Reserved

(20 July 2012) Bishal and Pooya - Created file.
*/

// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#include "parent_main.h"
#include "parent_frame.h"
IMPLEMENT_APP(ClientParentApp)
// the application class
// -----------------------------------------------
bool ClientParentApp::OnInit()
{
    // Create the main application window
    ParentFrame *frame = new ParentFrame();
    // Show it and tell the application that it's our main window
    frame->Show(true);
    SetTopWindow(frame);
    // success
    return true;
}
/**
 \file parent_frame.h
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
 \author (c)2012 Blekinge Institute of Technology . All
 rights Reserved
 (20 July 2012) Bishal and Pooya
 - Created file.
*/
// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef PARENTFRAME_H
#define PARENTFRAME_H
// Additional Dependencies
// -----------------------------------------------
// headers
// -----------------------------------------------
// For compilers that support precompilation, includes "wx/wx.h".
// include frame grabber class
#include "frame_grabber.h"
//opencv include files
#include <cv.h>
#include <highgui.h>
#include<wx/file.h>
#ifdef __BORLANDC__
#pragma hdrstop
#endif
// for all others, include the necessary headers
#ifndef WX_PRECOMP
    #include "wx/wx.h"
#endif
#include "wx/wxprec.h"
#include "wx/socket.h"
#include "wx/wfstream.h"
#include <wx/timer.h>
#include <wx/sizer.h>
#include <wx/valgen.h>
// Define a new frame type: this is going to be our main frame
class ParentFrame : public wxFrame
{
    public:
        ParentFrame();
        ~ParentFrame();
        // event handlers for File menu
void OnQuit(wxCommandEvent& event);
void OnAbout(wxCommandEvent& event);
// event handlers for Socket menu
void OnOpenConnection(wxCommandEvent& event);
void OnStartStreaming(wxCommandEvent& event);
void OnCloseConnection(wxCommandEvent& event);
// socket event handler
void OnSocketEvent(wxSocketEvent& event);
// convenience functions
void UpdateStatusBar();
// timer invokes this method to stream data, periodically
void GrabAndSend();
// OnTimer event handler
void OnTimer(wxTimerEvent & event);

public:

// ------------------------------------------
// constants
// ------------------------------------------
// IDs for the controls and the menu commands
enum
{
    // menu items
    CLIENT_QUIT = wxID_EXIT,
    CLIENT_ABOUT = wxID_ABOUT,
    CLIENT_OPEN = 100,
    CLIENT_STREAMING,
    CLIENT_CLOSE,
    CLIENT_TIMER,

    // id for socket
    SOCKET_ID
};

private:
wxSocketClient *m_sock;
wxTextCtrl *m_text;
wxMenu *m_menuFile;
wxMenu *m_menuStreaming;
wxMenuBar *m_menuBar;
bool m_busy;
// timer
wxStopWatch *test_timer;
wxFile *write_file;
int counter;
wxTimer *m_timer;
IplImage *m_image;
CvCapture *m_capture;
FrameGrabber *m_frameGrabber;

protected:
DECLARE_EVENT_TABLE()

};
#endif // PARENTFRAME_H

/**
 \file parent_frame.cpp
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
#include "parent_frame.h"

ParentFrame::ParentFrame() : wxFrame((wxFrame *)NULL, wxID_ANY, 
(_("Streaming : Client 1GS"), 
wxDefaultPosition, wxSize(400, 300))
{
    // Give the frame an icon
    SetIcon(wxICON(mondrian));
    // Make menus
    m_menuFile = new wxMenu();
    m_menuFile->Append(CLIENT_ABOUT, _("&About...\tCtrl-A"), _("Show about dialog"));
    m_menuFile->AppendSeparator();
    m_menuFile->Append(CLIENT_QUIT, _("E&xit\tAlt-X"), _("Quit client"));
    m_menuStreaming = new wxMenu();
    m_menuStreaming->Append(CLIENT_OPEN, _("&Open session"), _("Connect to server"));
    m_menuStreaming->AppendSeparator();
    m_menuStreaming->Append(CLIENT_STREAMING, _("Start live streaming"), _("Start transferring data"));
    m_menuStreaming->AppendSeparator();
    m_menuStreaming->Append(CLIENT_CLOSE, _("&Close session"), _("Close connection"));
    // Append menus to the menubar
    m_menuBar = new wxMenuBar();
    m_menuBar->Append(m_menuFile, _("File"));
    m_menuBar->Append(m_menuStreaming, _("Streaming"));
    SetMenuBar(m_menuBar);
    #if wxUSE_STATUSBAR
    // Status bar
    CreateStatusBar(2);
    #endif // wxUSE_STATUSBAR
    // Make a textctrl for logging
    m_text = new wxTextCtrl(this, wxID_ANY, 
(_("Welcome to Live 3D TV Streaming: Client 1GS\nClient 1GS ready!\n"),
wxDefaultPosition, wxDefaultSize,
wxTE_MULTILINE | wxTE_READONLY);
    // Create the socket
    m_sock = new wxSocketClient();
    // Setup the event handler and subscribe to most events
    m_sock->SetEventHandler(*this, SOCKET_ID);}
m_sock->SetNotify(wxSOCKET_CONNECTION_FLAG | wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);

m_sock->Notify(true);
// creating the instance of the timer to periodically send images to the server
m_timer = new wxTimer(this, CLIENT_TIMER);
// an instance to FrameGrab object must be created,
m_frameGrabber = new FrameGrab();
m_busy = false;
write_file = new wxFile("client1.txt", wxFile::write);
test_timer = new wxStopWatch;
counter = 0;
UpdateStatusBar();
}

BEGIN_EVENT_TABLE(ParentFrame, wxFrame)
  EVT_MENU(CLIENT_QUIT, ParentFrame::OnQuit)
  EVT_MENU(CLIENT_ABOUT, ParentFrame::OnAbout)
  EVT_MENU(CLIENT_OPEN, ParentFrame::OnOpenConnection)
  EVT_MENU(CLIENT_STREAMING, ParentFrame::OnStartStreaming)
  EVT_MENU(CLIENT_CLOSE, ParentFrame::OnCloseConnection)
  EVT_SOCKET(SOCKET_ID, ParentFrame::OnSocketEvent)
  EVT_TIMER(CLIENT_TIMER, ParentFrame::OnTimer)
END_EVENT_TABLE()
// end of event table
ParentFrame::~ParentFrame()
{
  // No delayed deletion here, as the frame is dying anyway
  delete m_sock;
}

void ParentFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
  // true is to force the frame to close
  m_frameGrabber->MilClose();
  Close(true);
}

void ParentFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{
  wxString str = wxString::Format( wxT("%d
"), x );
  write_file->Write(str);
  write_file->Write(str);
}

void ParentFrame::OnOpenConnection(wxCommandEvent& WXUNUSED(event))
{
  wxIPV4address addr;

  m_menuStreaming->Enable(CLIENT_OPEN, false);
  m_menuStreaming->Enable(CLIENT_CLOSE, false);

  // Ask user for server address
wxString hostname = wxGetTextFromUser(_("Enter the address of the server:"), _("Connect ...")), _("localhost"));

addr.Hostname(hostname);
addr.Service(3000);

// There are two ways to use Connect(): blocking and non-blocking, depending on the value passed as the 'wait' (2nd) parameter.
// Connect(addr, true) will wait until the connection completes, returning true on success and false on failure. This call blocks the GUI (this might be changed in future releases to honour the wxSOCKET_BLOCK flag).
// Connect(addr, false) will issue a nonblocking connection request and return immediately. If the return value is true, then the connection has been already successfully established. If it is false, you must wait for the request to complete, either with WaitOnConnect() or by watching wxSOCKET_CONNECTION / LOST events (please read the documentation).
// WaitOnConnect() itself never blocks the GUI (this might change in the future to honour the wxSOCKET_BLOCK flag). This call will return false on timeout, or true if the connection request completes, which in turn might mean:
// a) That the connection was successfully established
// b) That the connection request failed (for example, because it was refused by the peer.
// Use IsConnected() to distinguish between these two.
// So, in a brief, you should do one of the following things:
// For blocking Connect:
//   bool success = client->Connect(addr, true);
// For nonblocking Connect:
//   client->Connect(addr, false);
//   bool waitmore = true;
//   while (! client->WaitOnConnect(seconds, millis) && waitmore )
//   {
//     // possibly give some feedback to the user,
//     // update waitmore if needed.
//   }
//   bool success = client->IsConnected();
// And that's all :-)

m_text->AppendText(_("\nTrying to connect (timeout = 10 sec) ...\n"));
m_sock->Connect(addr, false);
m_sock->WaitOnConnect(10);
if (m_sock->IsConnected())
{
    m_text->AppendText(_("Succeeded ! Connection established\n"));
}
else
{
    m_sock->Close();
    m_text->AppendText(_("Failed ! Unable to connect\n"));
    wxMessageBox(_("Can't connect to the specified host"), _("Alert !"));
}
UpdateStatusBar();
}

void ParentFrame::OnStartStreaming(wxCommandEvent& WXUNUSED(event))
{
    // first lets initialize the MIL library and digitizer and start the timer
    m_frameGrabber->MilInit("I:/xxx_color.dcf");
    // here we start the timer so it sends images periodically,
    m_timer->Start(50);
    //test_timer->Start();
    m_image = NULL;
    //m_capture = cvCaptureFromFile("Knights Quest.wmv");
}

void ParentFrame::OnCloseConnection(wxCommandEvent& WXUNUSED(event))
{
    m_timer->Stop();
    m_frameGrabber->MilClose();
    m_sock->Close();
    UpdateStatusBar();
}

void ParentFrame::OnSocketEvent(wxSocketEvent& event)
{
    wxString s = _("OnSocketEvent: ");
    switch(event.GetSocketEvent())
    {
    case wxSOCKET_INPUT      : s.Append(_("wxSOCKET_INPUT\n"));
        break;
    case wxSOCKET_LOST       : s.Append(_("wxSOCKET_LOST\n")); break;
    case wxSOCKET_CONNECTION : s.Append(_("wxSOCKET_CONNECTION\n"));
        break;
    default                  : s.Append(_("Unexpected event !\n"));
        break;
    }
    m_text->AppendText(s);
    UpdateStatusBar();
}

// convenience functions
void ParentFrame::UpdateStatusBar()
{
    wxString s;
    if (!m_sock->IsConnected())
    {
        m_timer->Stop();
        s.Printf(_("Not connected"));
    }
else {
wxIPV4address addr;
    m_sock->GetPeer(addr);
    s.Printf(_("%s : %d"), (addr.Hostname()).c_str(),
addr.Service());
}
#endif // wxUSE_STATUSBAR
m_menuStreaming->Enable(CLIENT_OPEN, !m_sock->IsConnected() && !m_busy);
    m_menuStreaming->Enable(CLIENT_STREAMING, m_sock->IsConnected() && !m_busy);
    m_menuStreaming->Enable(CLIENT_CLOSE, m_sock->IsConnected());
}
void ParentFrame::OnTimer(wxTimerEvent & event)
{
counter++;
    wxString str;
    int x;

    // stopwatch to measure time intervals
    test_timer->Start();
    // this is where we get the images from the frame grabber
    //m_image = cvQueryFrame(m_capture); // instead of this it will be from mil?
    IplImage* green_right_image = NULL;
    IplImage* blue_left_image = NULL;
    m_frameGrabber->MilGetImages( M_CH0, green_right_image, blue_left_image );
    //cvCvtColor(m_image, m_image, CV_BGR2RGB);
    if(counter%15==0){
        x = test_timer->Time();
        str = wxString::Format( wxT("Query Image: %d ms
"), x );
        write_file->Write(str);
    }
    m_busy = true;
    UpdateStatusBar();
    // Tell the server which test we are running
    unsigned char c = 0xDE;
    m_sock->Write(&c, 1);
    // This test also is similar to the first one but it sends a
    // large buffer so that wxSocket is actually forced to split
    // it into pieces and take care of sending everything before
    // returning.
    if( blue_left_image->width == green_right_image->width && green_right_image->width<999 && green_right_image->height<999 )
    {
        m_sock->SetFlags(wxSOCKET_WAITALL);
        // First get the image width and height and send to the
        // server/client
        int height = green_right_image->height;
        int width = green_right_image->width;
        char c_height[3];
        char c_width[3];
        itoa(height, c_height, 10);
        itoa(width, c_width, 10);
test_timer->Start();
// send the width and height of the image first
m_sock->Write(c_height, 3);
m_sock->Write(c_width, 3);
// now write the images
m_sock->Write(blue_left_image->imageData, blue_left_image->imageSize);
m_sock->Write(green_right_image->imageData, green_right_image->imageSize);
// store timings here......
if(counter%15==0){
x = test_timer->Time();
str = wxString::Format( wxT("Writing Images on Socket: %d ms\n"), x );
write_file->Write(str);
}
// clear buffers
cvReleaseImage( &blue_left_image );
cvReleaseImage( &green_right_image );
m_busy = false;
UpdateStatusBar();
/**
 \file frame_grabber.h
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
 \author 2012 Blekinge Institute of Technology . All rights Reserved
 (18 May 2012) Bishal and Pooya
 - Created file.
*/
#ifndef FRAMEGRABBER_H
#define FRAMEGRABBER_H
#include <Mil.h>
#include "highgui.h"
using namespace cv;
#define CHECK_EXIT(x) {if(!(x)){printf("Error %s\n",#x);exit(0);}}
class FrameGrab
{
private:
  MIL_ID MilApplication,
       /* Application identifier. */
  MilSystem,
       /* System identifier. */
  //MilDisplay,
       /* Display identifier. */
  MilDigitizer,
       /* Digitizer identifier. */
  MilImageDisp,
       /* Image buffer identifier. */
  MilTempbuf;
  bool stopGrab;
public:
  bool MilGetImage( int , IplImage*& , IplImage*& , IplImage*& , IplImage*& , IplImage*& , int *,
                   int *);//displays video
  bool MilGetImages(int ,IplImage*& , IplImage*&);//ref two images
  bool MilInit(char *DCF_NAME);
  bool MilClose();
  void showImages();
  void stop_Grab(bool cmd) {stopGrab = cmd;}
};
#ifndef // FRAMEGRABBER_H
/**
 \file frame_grabber.cpp
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
 \author 2012 Blekinge Institute of Technology . All rights Reserved
 (18 May 2012) Bishal and Pooya
 - Created file.
 */
// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
#include "frame_grabber.h"
bool FrameGrab::MilGetImage(int Channel, IplImage * &img, IplImage * &rr, IplImage * &gg, IplImage * &bb, int *width, int *height)
{
    try
    {
        MbufClear(MilImageDisp, 0);
        MbufClear(MilTempbuf, 0);
        MdigChannel(MilDigitizer, Channel);
        //MdigControl(MilDigitizer, M_GRAB_TRIGGER, M_ACTIVATE );
        MdigGrab(MilDigitizer, MillImageDisp);
        int w=MbufInquire(MilImageDisp,M_SIZE_X,M_NULL);;
        int h=MbufInquire(MilImageDisp,M_SIZE_Y,M_NULL);;
        int band=MbufInquire(MilImageDisp,M_SIZE_BAND,M_NULL);;
        *width = w;
        *height = h;
        int pitch=MbufInquire(MilImageDisp,M_PITCH_BYTE,M_NULL);;
        if(!rr)
            rr=cvCreateImage(cvSize(w,h),8,1);,
        if(!gg)
            gg=cvCreateImage(cvSize(w,h),8,1);
        if(!bb)
            bb=cvCreateImage(cvSize(w,h),8,1);
        if(!img)
            img=cvCreateImage(cvSize(w,h),8,band);
        MbufGet(MilImageDisp, img->imageData);
        img->origin = IPL_ORIGIN_TL;
        // get red
        MbufCopyColor(MilImageDisp,MilTempbuf, M_RED );
        MbufGet(MilTempbuf, rr->imageData);
        rr->origin = IPL_ORIGIN_TL;
        MbufCopyColor(MilImageDisp, MilTempbuf, M_BLUE );
        MbufGet(MilTempbuf, bb->imageData);
        bb->origin = IPL_ORIGIN_TL;
        //get the green band
        MbufClear(MilTempbuf, 0);
        MbufCopyColor(MilImageDisp,MilTempbuf, M_GREEN );
        MbufGet(MilTempbuf, gg->imageData);
        gg->origin = IPL_ORIGIN_TL;
    }
    catch(void*)
    {
        return false;
    }
    return true;
};
bool FrameGrab::MilGetImages( int Channel, IplImage * &gg, IplImage * &bb )
{
    try
    {
        MbufClear(MilImageDisp, 0);
        MbufClear(MilTempbuf, 0);
        MdigChannel(MilDigitizer, Channel);
        //MdigControl(MilDigitizer, M_GRAB_TRIGGER, M_ACTIVATE );
        MdigGrab(MilDigitizer, MilImageDisp);

        int w=MbufInquire(MilImageDisp,M_SIZE_X,M_NULL);
        int h=MbufInquire(MilImageDisp,M_SIZE_Y,M_NULL);
        int band=MbufInquire(MilImageDisp,M_SIZE_BAND,M_NULL);
        int pitch=MbufInquire(MilImageDisp,M_PITCH_BYTE,M_NULL);
        if(!gg)
            gg=cvCreateImage(cvSize(w,h),8,1);
        if(!bb)
            bb=cvCreateImage(cvSize(w,h),8,1);
        // get the blue band
        MbufCopyColor(MilImageDisp, MilTempbuf, M_BLUE );
        MbufGet(MilTempbuf, bb->imageData);
        bb->origin = IPL_ORIGIN_TL;
        //get the green band
        MbufClear(MilTempbuf, 0);
        MbufCopyColor(MilImageDisp,MilTempbuf, M_GREEN );
        MbufGet(MilTempbuf, gg->imageData);
    }
    catch(void*)
    {
        return false;
    }
    return true;
}

bool FrameGrab::MilInit(char *DCF_NAME)
{
    stopGrab = false;
    try
    {
        MappAlloc(M_DEFAULT, &MilApplication);
        MsysAlloc(M_SYSTEM_METEOR_II, M_DEF_SYSTEM_NUM, M_SETUP, &MilSystem);
        //MdispAlloc(MilSystem, M_DEFAULT, M_DEF_DISPLAY_FORMAT, M_DEFAULT, &MilDisplay);
        MdigAlloc(MilSystem, M_DEFAULT, DCF_NAME, M_DEFAULT, &MilDigitizer);
        //MdigChannel(MilDigitizer, M_CH2);
        MbufAllocColor(MilSystem,
                       MdigInquire(MilDigitizer, M_SIZE_BAND, M_NULL),
                       (long) (MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)),
                       (long) (MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)),
                       8L+M_UNSIGNED,
                       M_IMAGE+M_GRAB+M_DISP,
                       &MilImageDisp);
        MbufAlloc2d(MilSystem,
                    (long) (MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)),
                    (long) (MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)),
                    8L+M_UNSIGNED,
                    M_IMAGE+M_GRAB+M_DISP,
                    &MilTempbuf);
    }
MdigControl(MilDigitizer, M_GRAB_TRIGGER, M_ENABLE); // working better with analog mode
MdigControl(MilDigitizer, M_GRAB_FIELD_NUM, 2);
MdigControl(MilDigitizer, M_GRAB_START_MODE, M_FIELD_START);
MbufClear(MilImageDisp, 0);

catch(void*){
    return false;
}
return true;

bool FrameGrab::MilClose() {
    try{
        MbufFree(MilImageDisp);
        MbufFree(MilTempbuf);
        MdigFree(MilDigitizer);
        MsysFree(MilSystem);
        MappFree(MilApplication);
    }
    catch (void*){
        return false;
    }
    return true;
}
void FrameGrab::showImages() {
    IplImage *img0 = 0;
    IplImage *img1 = 0;
    IplImage *r = 0;
    IplImage *g = 0;
    IplImage *b = 0;
    IplImage *sideBySideImage = 0;
    int Image_id = 0;
    int width = 0;
    int height = 0;
    do{
        MilGetImage(M_CH0, img0, r, g, b, &width, &height);
        CHECK_EXIT(g);
        CHECK_EXIT(b);
        CHECK_EXIT(img0);
        // putting two images side by side
        if (!sideBySideImage)
            sideBySideImage = cvCreateImage(cvSize(width*2, height), IPL_DEPTH_8U, 1);
        cvSetImageROI(sideBySideImage, cvRect(0*width, 0, width, height));
        cvCopy(g, sideBySideImage);
        cvSetImageROI(sideBySideImage, cvRect(1*width, 0, width, height));
        cvCopy(b, sideBySideImage);
        cvResetImageROI(sideBySideImage);
        int c = cvWaitKey(1);
        if (c == 'g')
            {
                char img_name[100];
}
cvSaveImage(img_name, sideBySideImage);
Image_id++;
}
else if (c == 27)
    break;
    cvShowImage( "Green", g );
    cvShowImage( "Blue", b );
}
while (!stopGrab);
if ( stopGrab )
{
    cvReleaseImage(&img0);
    cvReleaseImage(&img1);
    cvReleaseImage(&r);
    cvReleaseImage(&g);
    cvReleaseImage(&b);
    cvDestroyAllWindows();
}

Server Application

/**
 * file
 * parent_main.h
 * author
 * Bishal Neupane
 * author
 * Pooya Moazzeni Bikani
 * author
 * 2012 Blekinge Institute of Technology. All rights Reserved
 * (20 July 2012)
 * - Bishal and Pooya
 * 
 * Created file.
 */
// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef __PARENT_MAIN_H__
#define __PARENT_MAIN_H__
// 
// the application class
// For compilers that support precompilation, includes "wx/wx.h".
#include "wx/wxprec.h"
#include "wx/socket.h"
#ifdef __BORLANDC__
#pragma hdrstop
#endif
// for all others, include the necessary headers
#ifndef WX_PRECOMP
    #include "wx/wx.h"
#endif
// -----------------------------

49
// resources
// -----------------------------------------------
// the application icon
#if defined(__WXGTK__) || defined(__WXX11__) || defined(__WXMOTIF__) || defined(__WXMAC__)
#include "mondrian.xpm"
#endif
// Define a new application type
// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
class ParentApp : public wxApp {
public:
   virtual bool OnInit();
   //bool OnExit();
};
#endif __PARENT_MAIN_H__

/**
 \file parent_main.cpp
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
 \author (c)2012 Blekinge Institute of Technology . All rights Reserved
 (20 July 2012) Bishal and Pooya
 - Created file.
 */
// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#include "parent_main.h"
#include "parent_frame.h"
IMPLEMENT_APP(ParentApp)
bool ParentApp::OnInit()
{
   // Create the main application window
   MyFrame *frame = new MyFrame();
   // Show it and tell the application that it's our main window
   frame->Show(true);
   SetTopWindow(frame);
   // Success
   return true;
}
/**
 \file parent_frame.h
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
 \author (c)2012 Blekinge Institute of Technology . All rights Reserved
 (20 July 2012) Bishal and Pooya
 - Created file.
 */
// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef PARENTFRAME_H
#define PARENTFRAME_H
// ---------------
// headers
// For compilers that support precompilation, includes "wx/wx.h".
#include "wx/wxprec.h"
//opencv include files
#include <cv.h>
#include <highgui.h>
#include <wx/timer.h>
#include <wx/file.h>
#ifdef __BORLANDC__
     #pragma hdrstop
#endif
// for all others, include the necessary headers
#ifndef WX_PRECOMP
     #include "wx/wx.h"
#endif
#include "wx/socket.h"

// Define a new frame type: this is going to be our main frame

class MyFrame : public wxFrame
{
public:
    MyFrame();
    ~MyFrame();
    // event handlers (these functions should _not_ be virtual)
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
    void OnServerEvent(wxSocketEvent& event);
    void OnSocketEvent(wxSocketEvent& event);
    void OnStreamVideo(wxSocketBase *sock);
    // convenience functions
    void UpdateStatusBar();

public:
    // IDs for the controls and the menu commands
    enum
    {
        // menu items
        SERVER_QUIT = wxID_EXIT,
        SERVER_ABOUT = wxID_ABOUT,
        // id for sockets
        SERVER_ID = 100,
        SOCKET_ID
    };

private:
    wxSocketBase *m_client1;
    wxSocketBase *m_client2;
    wxSocketServer *m_server;
    wxTextCtrl *m_text;
    wxMenu *m_menuFile;
    wxMenuBar *m_menuBar;
    bool m_busy;
    int m_numClients;
    // timer
    wxStopWatch *test_timer;
    wxFile *write_file;
    int counter;

    // any class wishing to process wxWidgets events must use this macro
    DECLARE_EVENT_TABLE()
};
#endif // PARENTFRAME_H
/**
 \file parent_frame.cpp
 \author Bishal Neupa
 \author Pooya Moazzeni Bikani
 \author (c)2012 Blekinge Institute of Technology . All
 rights Reserved
 (20 July 2012) Bishal and Pooya
 - Created file.
 */

#include "parent_frame.h"

// event tables and other macros for wxWidgets
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
EVT_MENU(SERVER_QUIT, MyFrame::OnQuit)
EVT_MENU(SERVER_ABOUT, MyFrame::OnAbout)
EVT_SOCKET(SERVER_ID, MyFrame::OnServerEvent)
EVT_SOCKET(SOCKET_ID, MyFrame::OnSocketEvent)
END_EVENT_TABLE()

// main frame // frame constructor
MyFrame::MyFrame() : wxFrame((wxFrame *)NULL, wxID_ANY,
_("Streaming: Server"),
wxDefaultPosition, wxSize(400, 300))
{
  // Give the frame an icon
  SetIcon(wxICON(mondrian));
  // Make menus
  m_menuFile = new wxMenu();
  m_menuFile->Append(SERVER_ABOUT, _("&About...	Ctrl\-A"),
  _("Show about dialog"));
  m_menuFile->AppendSeparator();
  m_menuFile->Append(SERVER_QUIT, _("E\&xit	Alt\-X"), _("Quit server"));
  // Append menus to the menubar
  m_menuBar = new wxMenuBar();
  m_menuBar->Append(m_menuFile, _("&File"));
  SetMenuBar(m_menuBar);
#if wxUSE_STATUSBAR
  // Status bar
  CreateStatusBar(2);
#endif // wxUSE_STATUSBAR
  // Make a textctrl for logging
  m_text = new wxTextCtrl(this, wxID_ANY,
  _("Welcome to Live 3D TV Streaming: SERVER\n"),
  wxDefaultPosition, wxDefaultSize,
  wxT_MULTILINE | wxT_READONLY);
  // Create the address - defaults to localhost:0 initially
  wxIPV4address addr;
  addr.Service(3000);
  // Create the socket
  m_server = new wxSocketServer(addr);
  // We use Ok() here to see if the server is really listening
  if (! m_server->Ok())

m_text->AppendText(_("Could not listen at the specified port
\n\n"));
    return;
}  
else  
{  
    m_text->AppendText(_("Server listening.\n\n"));
}
// Setup the event handler and subscribe to connection events
m_server->SetEventHandler(*this, SERVER_ID);
m_server->SetNotify(wxSOCKET_CONNECTION_FLAG);
m_server->Notify(true);
m_client1 = NULL;
m_client2 = NULL;
m_busy = false;
m_numClients = 0;
write_file = new wxFile("server.txt", wxFile::write);
test_timer = new wxStopWatch;
counter = 0;
UpdateStatusBar();
}  
MyFrame::~MyFrame()  
{
    // No delayed deletion here, as the frame is dying anyway
    delete m_server;
}
// event handlers
void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))  
{
    // true is to force the frame to close
    // m_liveView->Stop();
    Close(true);
}
void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))  
{
    wxMessageBox(_("Live 3D TV Streaming: SERVER - IPL Lab\n\n(c) 2012 Bishal Neupane, Pooya Moazzeni\n"),
               _("About Application"),
               wxOK | wxICON_INFORMATION, this);
}
void MyFrame::OnServerEvent(wxSocketEvent& event)  
{
    wxString s = _("OnServerEvent: ");
    wxSocketBase *sock;
    switch(event.GetSocketEvent())  
    {  
        case wxSOCKET_CONNECTION : s.Append(_("wxSOCKET_CONNECTION\n")); break;
        default : s.Append(_("Unexpected event !\n")); break;
    }
    m_text->AppendText(s);
    // Accept new connection if there is one in the pending
    // connections queue, else exit. We use Accept(false) for
    // non-blocking accept (although if we got here, there
    // should ALWAYS be a pending connection).
    sock = m_server->Accept(false);
    if (sock)  
    {

m_text->AppendText(_("New client connection accepted\n\n"));
}
else
{
    m_text->AppendText(_("Error: couldn't accept a new connection\n\n"));
    return;
}
soc->SetEventHandler(*this, SOCKET_ID);
soc->SetNotify(wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);
soc->Notify(true);
m_numClients++;
if( m_numClients == 1 )
    m_client1 = soc;
else if( m_numClients == 2 )
    m_client2 = soc;
UpdateStatusBar();
}

void MyFrame::OnSocketEvent(wxSocketEvent& event)
{
    wxString s = _("OnSocketEvent: ");
    wxSocketBase *sock = event.GetSocket();
    // First, print a message
    switch(event.GetSocketEvent())
    {
    case wxSOCKET_INPUT:
        s.Append(_("wxSOCKET_INPUT\n"));
        break;
    case wxSOCKET_LOST:
        s.Append(_("wxSOCKET_LOST\n"));
        break;
    default:
        s.Append(_("Unexpected event !\n"));
        break;
    }
    m_text->AppendText(s);
    // Now we process the event
    switch(event.GetSocketEvent())
    {
    case wxSOCKET_INPUT:
    {
        // We disable input events, so that the test doesn't trigger
        // wxSocketEvent again.
        sock->SetNotify(wxSOCKET_LOST_FLAG);
        // Which test are we going to run?
        unsigned char c;
        sock->Read(&c, 1);
        switch (c)
        {
        case 0xDE: OnStreamVideo(sock); break;
        default:
            m_text->AppendText(_("Unknown test id received from client\n\n"));
        }
        // Enable input events again.
        sock->SetNotify(wxSOCKET_LOST_FLAG | wxSOCKET_INPUT_FLAG);
        break;
    }
    case wxSOCKET_LOST:
    {
        m_numClients--;
        // Destroy() should be used instead of delete wherever possible,
        // due to the fact that wxSocket uses 'delayed events' (see the
        // documentation for wxPostEvent) and we don't want an event to
        // arrive to the event handler (the frame, here) after the socket
// has been deleted. Also, we might be doing some other thing with
// the socket at the same time; for example, we might be in the
// middle of a test or something. Destroy() takes care of all
// this for us.
m_text->AppendText(_("Deleting socket.\n\n"));
sock->Destroy();
break;
}
default: ;
}
UpdateStatusBar();

// convenience functions
void MyFrame::UpdateStatusBar()
{
#if wxUSE_STATUSBAR
wxString s;
s.Printf(_("%d clients connected"), m_numClients);
setStatusText(s, 1);
#endif // wxUSE_STATUSBAR
}
void MyFrame::OnStreamVideo(wxSocketBase *sock)
{
char *c_height;
char *c_width;
counter++;
wxString str;
int x;
IplImage *image_left = NULL;
IplImage *image_right = NULL;
char *sockdata1;
char *sockdata2;
// This test is similar to the first one, but the len is
// expressed in kbytes - this tests large data transfers.
sock->SetFlags(wxSOCKET_WAITALL);
c_height = new char[3];
c_width = new char[3];
// Read the size
sock->Read(c_height, 3);
sock->Read(c_width, 3);
if (m_client2 )
{
    m_client2->Write(c_height, 3);
    m_client2->Write(c_width, 3);
}
int hh = atoi( c_height );
int ww = atoi( c_width );
image_left = cvCreateImage( cvSize(ww, hh), IPL_DEPTH_8U, 1);
image_right = cvCreateImage( cvSize(ww, hh), IPL_DEPTH_8U, 1);
cvZero(image_left);
cvZero(image_right);
sockdata1 = new char[image_left->imageSize];
sockdata2 = new char[image_right->imageSize];
test_timer->Start();
sock->Read(sockdata1, image_left->imageSize);
sock->Read(sockdata2, image_right->imageSize);
if(counter%15==0){
x = test_timer->Time();
str = wxString::Format( wxT("Read Image: %d ms\n"), x );
write_file->Write(str);
}
test_timer->Start();
if( m_client2 )
    m_client2->Write(sockdata1, image_left->imageSize );
if( m_client2 )
    m_client2->Write(sockdata2, image_right->imageSize );
if(counter%15==0){
    x = test_timer->Time();
    str = wxString::Format( wxT("Write Image: %d ms\n"), x );
    write_file->Write(str);
}
// free the resources
delete [] sockdata1;
delete [] sockdata2;
delete [] c_height;
delete [] c_width;
cvReleaseImage( &image_left );
cvReleaseImage( &image_right );

Client 2 Application

/**
 * file parent_main.h
 * author Bishal Neupane
 * author Pooya Moazzeni Bikani
 * author 2012 Blekinge Institute of Technology. All rights Reserved
 */
#endif __PARENT_MAIN_H__
#define __PARENT_MAIN_H__

// the application class
// For compilers that support precompilation, includes "wx/wx.h".
#include "wx/wxprec.h"
#include "wx/socket.h"
#ifdef __BORLANDC__
    #pragma hdrstop
#endif
// for all others, include the necessary headers
#ifndef WX_PRECOMP
    #include "wx/wx.h"
#endif

// resources

// the application icon
#endif

class ParentApp : public wxApp
{
public:
    virtual bool OnInit();
    //bool OnExit();
};
#endif __PARENT_MAIN_H__

/**
\file parent_main.cpp
\author Bishal Neupane
\author Pooya Moazzeni Bikani
\author 2012 Blekinge Institute of Technology. All rights Reserved
(20 July 2012) Bishal and Pooya
- Created file.
*/

bool ParentApp::OnInit()
{
    // Create the main application window
    MyFrame *frame = new MyFrame();
    // Show it and tell the application that it's our main window
    frame->Show(true);
    SetTopWindow(frame);
    // Success
    return true;
}

/**
\file parent_frame.h
\author Bishal Neupane
\author Pooya Moazzeni Bikani
*/
 ifndef PARENTFRAME_H
#define PARENTFRAME_H
#include "LiveViewFrame.h"
#include "GLCanvasTimer.h"

#include "wx/wxprec.h"
#include <cv.h>
#include <highgui.h>
#ifdef __BORLANDC__
  #pragma hdrstop
#endif

// For all others, include the necessary headers
#ifndef WX_PRECOMP
  # include "wx/wx.h"
#endif
#include "wx/socket.h"

// Define a new frame type: this is going to be our main frame

class MyFrame : public wxFrame
{
private:
  // The live view frame
  LiveViewFrame* m_liveView;
  // timer for rendering,
  GLCanvasTimer *m_timer;

public:
  MyFrame();
  ~MyFrame();
  void OnQuit(wxCommandEvent& event);
  void OnAbout(wxCommandEvent& event);
  void OnSocketEvent(wxSocketEvent& event);
  void OnIncomingData(wxSocketBase *sock);
  void UpdateStatusBar();

public:
  // IDs for the controls and the menu commands
  enum
  {
    // menu items
    SERVER_QUIT = wxID_EXIT,
    SERVER_ABOUT = wxID_ABOUT,
CLIENT_OPEN,
CLIENT_CLOSE,

// id for sockets
SERVER_ID = 100,
SOCKET_ID
};

private:
wxSocketClient *m_sock;
wxTextCtrl     *m_text;
wxMenu         *m_menuFile;
wxMenu         *m_menuStreaming;
wxMenuBar      *m_menuBar;
bool           m_busy;
int            m_numClients;
//test data
wxStopWatch    *test_timer;
xwFile         *write_file;
int            counter;

// any class wishing to process wxWidgets events must use this macro
protected:
DECLARE_EVENT_TABLE()
};
#endif // PARENTFRAME_H

/**
 \file     parent_frame.cpp
 \author   Bishal Neupane
 \author   Pooya Moazzeni Bikani
 \author   (c)2012 Blekinge Institute of Technology . All
 rights Reserved
 (20 July 2012)     Bishal and Pooya
 - Created file.
 */

#include "parent_frame.h"

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
  EVT_MENU(SERVER_QUIT, MyFrame::OnQuit)
  EVT_MENU(SERVER_ABOUT, MyFrame::OnAbout)
  EVT_MENU(CLIENT_OPEN,     MyFrame::OnOpenConnection)
  EVT_MENU(CLIENT_CLOSE,    MyFrame::OnCloseConnection)
  EVT_SOCKET(SOCKET_ID,  MyFrame::OnSocketEvent)
END_EVENT_TABLE()

MyFrame::MyFrame() : wxFrame((wxFrame *)NULL, wxID_ANY,
    _("Streaming: Client 2RR(Display)"),
    wxDefaultPosition, wxSize(400, 300))
{
    // Give the frame an icon
    SetIcon(wxICON(mondrian));
    // Make menus
    m_menuFile = new wxMenu();


m_menuFile->Append(SERVER_ABOUT, _("&About...\tCtrl-A"), _("Show about dialog"));
m_menuFile->AppendSeparator();
m_menuFile->Append(SERVER_QUIT, _("E&xit\tAlt-X"), _("Quit server"));
m_menuStreaming = new wxMenu();
m_menuStreaming->Append(CLIENT_OPEN, _("&Open session"), _("Connect to server"));
m_menuStreaming->AppendSeparator();
m_menuStreaming->Append(CLIENT_CLOSE, _("&Close session"), _("Close connection"));

// Append menus to the menubar
m_menuBar = new wxMenuBar();
m_menuBar->Append(m_menuFile, _("&File"));
m_menuBar->Append(m_menuStreaming, _("&Streaming"));
SetMenuBar(m_menuBar);

#if wxUSE_STATUSBAR
// Status bar
CreateStatusBar(2);
#endif // wxUSE_STATUSBAR

// Make a textctrl for logging
m_text = new wxTextCtrl(this, wxID_ANY, _("Welcome
to Live 3D TV Streaming: Client 2RR(Displ\n"),
wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE | wxTE_READONLY);

wxDefaultPosition, wxDefaultSize,
wxTE_MULTILINE | wxTE_READONLY);

// Create the socket
m_sock = new wxSocketClient();
// Setup the event handler and subscribe to most events
m_sock->SetEventHandler(*this, SOCKET_ID);
wxSOCKET_INPUT_FLAG |
wxSOCKET_CONNECTION_FLAG |
m_sock->Notify(true);
m_busy = false;
// test data
write_file = new wxFile("client2.txt", wxFile::write );
test_timer = new wxStopWatch;
counter = 0;
UpdateStatusBar();
}
MyFrame::~MyFrame()
{
    // No delayed deletion here, as the frame is dying anyway
    delete m_sock;
}

// event handlers
void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    // true is to force the frame to close
    m_liveView->Stop();
    Close(true);
}
void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{
    wxMessageBox(_("Live 3D TV Streaming: Client 2RR - IPL Lab\n(c) 2012 Bishal Neupane, Pooya Moazzeni\n"),

void MyFrame::OnOpenConnection(wxCommandEvent& WXUNUSED(event))
{
    wxIPV4address addr;
    m_menuStreaming->Enable(CLIENT_OPEN, false);
    m_menuStreaming->Enable(CLIENT_CLOSE, false);
    // Ask user for server address
    wxString hostname = wxGetTextFromUser(
        _("Enter the address of the server:'"),
        _("Connect ..."),
        _("localhost"));
    addr.Hostname(hostname);
    addr.Service(3000);
    m_text->AppendText(_("Trying to connect (timeout = 10 sec) ...\n"));
    m_sock->Connect(addr, false);
    m_sock->WaitOnConnect(10);
    if (m_sock->IsConnected())
    {
        m_text->AppendText(_("Succeeded ! Connection established\n"));
        // initialize the liveview and glcanvas to render the incoming
        data
        m_liveView = LiveViewFrame::Create(NULL);
        m_timer = new GLCanvasTimer(m_liveView);
        m_timer->Start();
    }
    else
    {
        m_sock->Close();
        m_text->AppendText(_("Failed ! Unable to connect\n"));
        wxMessageBox(_("Can't connect to the specified host"), _("Alert
"));
    }
    UpdateStatusBar();
}

void MyFrame::OnCloseConnection(wxCommandEvent& WXUNUSED(event))
{
    // destroy the frame
    m_liveView->Stop();
    m_sock->Close();
    UpdateStatusBar();
}

void MyFrame::OnSocketEvent(wxSocketEvent& event)
{
    wxString s = _("OnSocketEvent: ");
    wxSocketBase *sock = event.GetSocket();

    // First, print a message
    switch(event.GetSocketEvent())
    {
    case wxSOCKET_INPUT : s.Append(_("wxSOCKET_INPUT\n")); break;
    case wxSOCKET_LOST : s.Append(_("wxSOCKET_LOST\n")); break;
    case wxSOCKET_CONNECTION : s.Append(_("wxSOCKET_CONNECTION\n")); break;
    default : s.Append(_("Unexpected event !\n")); break;
    }
m_text->AppendText(s);

// Now we process the event
switch(event.GetSocketEvent())
{
    case wxSOCKET_INPUT:
    {
        // We disable input events, so that the test doesn't trigger
        // wxSocketEvent again.
        sock->SetNotify(wxSOCKET_LOST_FLAG);

        // Read the data that server has sent
        OnIncomingData( sock );

        // Enable input events again.
        sock->SetNotify(wxSOCKET_LOST_FLAG | wxSOCKET_INPUT_FLAG);
        break;
    }

    default: ;
}

UpdateStatusBar();
}

void MyFrame::OnIncomingData(wxSocketBase *sock)
{
    char *c_height;
    char *c_width;
    counter++;
    wxString str;
    int x;
    IplImage *image_left = NULL;
    IplImage *image_right = NULL;
    char *sockdata1;
    char *sockdata2;
    // This test is similar to the first one, but the len is
    // expressed in kbytes - this tests large data transfers.
    sock->SetFlags(wxSOCKET_WAITALL);
    c_height = new char[3];
    c_width = new char[3];
    // Read the size
    sock->Read(c_height, 3);
    sock->Read(c_width, 3);
    int hh = atoi( c_height );
    int ww = atoi( c_width );
    image_left = cvCreateImage( cvSize(ww, hh), IPL_DEPTH_8U, 1);
    image_right = cvCreateImage( cvSize(ww, hh), IPL_DEPTH_8U, 1);
    cvZero(image_left);
    cvZero(image_right);
    sockdata1 = new char[image_left->imageSize];
    sockdata2 = new char[image_right->imageSize];
    test_timer->Start();
    sock->Read(sockdata1, image_left->imageSize);
    sock->Read(sockdata2, image_right->imageSize);
    if(counter%15==0){
        x = test_timer->Time();
        str = wxString::Format( wxT("Read Images: %d ms\n"), x );
    }
}
write_file->Write(str);
}
image_left->imageData = sockdata1;
image_right->imageData = sockdata2;
// image_left and image_right must be 3 channel image after this point
if( !m_liveView->m_RENDER && m_liveView->m_READ )
{
    m_liveView->m_image1 = cvCreateImage(cvSize(image_left->width, image_left->height), IPL_DEPTH_8U, 3);
    m_liveView->m_image2 = cvCreateImage(cvSize(image_left->width, image_left->height), IPL_DEPTH_8U, 3);
    cvCvtColor(image_left, m_liveView->m_image1, CV_GRAY2RGB);
    cvCvtColor(image_right, m_liveView->m_image2, CV_GRAY2RGB);
    m_liveView->m_RENDER = true;
    m_liveView->m_READ = false;
}
delete [] sockdata1;
delete [] sockdata2;
delete [] c_height;
delete [] c_width;
cvReleaseImage( &image_left );
cvReleaseImage( &image_right );
}
// convenience functions
void MyFrame::UpdateStatusBar()
{
    wxString s;
    if (!m_sock->IsConnected())
    {
        s.Printf(_("Not connected"));
    }
    else
    {
        wxIPv4address addr;
        m_sock->GetPeer(addr);
        s.Printf(_("%s : %d"), (addr.Hostname()).c_str(),
        addr.Service());
    }
#if wxUSE_STATUSBAR
    SetStatusText(s, 1);
#endif // wxUSE_STATUSBAR
    m_menuStreaming->Enable(CLIENT_OPEN, !m_sock->IsConnected() && !m_busy);
    m_menuStreaming->Enable(CLIENT_CLOSE, m_sock->IsConnected());
}
/**
 * GLCanvasTimer.h
 * author Bishal Neupane
 * author Pooya Moazzeni Bikani
 * author 2012 Blekinge Institute of Technology. All rights Reserved
 * (25 May 2012) - Bishal and Pooya
 */
#include <wx/timer.h>
#include <wx/sizer.h>
class GLCanvasTimer : public wxTimer
{
private:
   //LiveViewGLCanvas *m_canvas;
   LiveViewFrame* m_liveView;
public:
   //GLCanvasTimer(LiveViewGLCanvas *m_canvas);
   GLCanvasTimer(LiveViewFrame *m_liveView);
   void Notify();
   void Start();
};

/**
file     GLCanvasTimer.cpp
author   Bishal Neupane
author   Pooya Moazzeni Bikani
author   2012 Blekinge Institute of Technology . All rights Reserved
(25 May 2012)  Bishal and Pooya
- Created file.
*/
// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#include "GLCanvasTimer.h"
//----------------------------------------------------------------------------------------
GLCanvasTimer::GLCanvasTimer(LiveViewFrame *m_liveView) : wxTimer()
{
   GLCanvasTimer::m_liveView = m_liveView;
}
void GLCanvasTimer::Notify()
{
   //SetData();
   m_liveView->SetData();
}
void GLCanvasTimer::Start()
{
   wxTimer::Start(50);
}
/**
file     LiveView_GLCanvas.h
author   Bishal Neupane
author   Pooya Moazzeni Bikani
author   2012 Blekinge Institute of Technology . All rights Reserved
(18 May 2012)  Bishal and Pooya
- Created file.
*/
// :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef __X_LIVEVIEW_GLCANVAS_H__
#define __X_LIVEVIEW_GLCANVAS_H__
// wx includes
//#include "frame_grabber.h"
#include <wx/glcanvas.h>
#include <wx/wx.h>
#include <sf_ilace_ogl.h>
#include <cv.h>
#include <higui.h>
#include<wx/file.h>
#include <wx/timer.h>

using namespace cv;

class LiveViewGLCanvas: public wxGLCanvas
{
    friend class CvLiveViewFrame;

public:
    //! Ctor
    LiveViewGLCanvas( wxWindow * parent,
                     wxWindowID id = wxID_ANY,
                     const wxPoint & pos = wxDefaultPosition,
                     const wxSize & size = wxDefaultSize,
                     long style = 0,
                     const wxString& name = _T("LiveViewGLCanvas") );

    //! Ctor
    LiveViewGLCanvas( wxWindow * parent,
                     const LiveViewGLCanvas * other,
                     wxWindowID id = wxID_ANY,
                     const wxPoint & pos = wxDefaultPosition,
                     const wxSize & size = wxDefaultSize,
                     long style = 0,
                     const wxString& name = _T("LiveViewGLCanvas") );

    //! Dtor
    ~LiveViewGLCanvas();

    bool SetTextures(GLuint* textures);
    void SetImgDims(int width, int height);
    void SetSeeFrontRes(bool val);
    bool IsSeeFrontModeAvailable();

    public:
        //! Paint events
    void OnPaint(wxPaintEvent& event);
        //! Size events
    void OnSize(wxSizeEvent& event);
        //! Refresh
    void OnEraseBackground(wxEraseEvent& event);
        //! Key press
    void OnKeyPress(wxKeyEvent& event);
        //! Mouse events
    void OnMouseEvent(wxMouseEvent& event);
    void trackerUpdate(double x, double y, double z);

    // testing texture
void drawTexture();
void drawTexture2();

public:
    //! Rendering modes
    enum RenderMode
    {
        eSXS = 1, // side-by-side
        eSF,
    };///enum RenderMode
    //! Set rendering type
    void SetRendering(RenderMode mode = eSXS);
    int GetRenderMode();
    bool MIL_buf_free( bool);
    bool stop;
    void setStop(){ stop = true; }
    int count_count; //

private:

    bool SetupRenderParams();
    //! Create an interlacer instance for the Seefront
    bool CreateSFInterlacer();
    //! Renderer for SeeFront
    void RenderSF( wxDC &);
    //! Renderer for side-by-side mode
    void RenderSxS( wxDC &);
    //! Required inits
    void InitGL();
    //! Check if OpenGL has been init'ed
    void checkGLInit();
    // set texture images by calling frame grabber
    void getTextures();
    //
    void paintNow();

public:

IplImage *ipl_image3;
IplImage *ipl_image4;
IplImage *ipl_image5;
IplImage *ipl_image6;

// timer
wxStopWatch *test_timer2;
wxFile *write_file2;
int counter2;

private:

IplImage *ipl_image1; // two iplimages
IplImage *ipl_image2;
GLuint m_gllist;
GLuint m_Images[2];
GLuint m_testImages[2];

int m_imagewidth;
int m_imageheight;

RenderMode m_iRenderMode;
bool m_init;
bool m_bSFReqToInit;
bool m_bSeefrontResAvailable;
bool m_bSeefrontAvailable;
bool m_bSeefrontCanInit;
```cpp
bool m_bSeefrontModeCheck;
bool m_init_mil_file;
sfogl::IlaceHandle m_interlacer;
DECLARE_EVENT_TABLE()
};
//end class LiveViewGLCanvas
#endif //__X_LIVEVIEW_GLCANVAS_H__
/**
 \file LiveView_GLCanvas.cpp
 \author Bishal Neupane
 \author Pooya Moazzeni Bikani
 \author 2012 Blekinge Institute of Technology . All rights Reserved
 (18 May 2012) Bishal and Pooya - Created file. */

// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#include "LiveView_GLCanvas.h"
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
void g_tracker_update(double x, double y, double z, void* user_data) {
    LiveViewGLCanvas* canvas = reinterpret_cast<LiveViewGLCanvas*>(user_data);
    if (canvas){
        canvas->trackerUpdate(x, y, z);
    }
}
//-------------------------------------------------------------------------------
//********** EVENTS ********://BEGIN_EVENT_TABLE(LiveViewGLCanvas, wxGLCanvas)
    EVT_SIZE(LiveViewGLCanvas::OnSize)
    EVT_PAINT(LiveViewGLCanvas::OnPaint)
    EVT_ERASE_BACKGROUND(LiveViewGLCanvas::OnEraseBackground)
    EVT_CHAR(LiveViewGLCanvas::OnKeyPress)
    EVT_MOUSE_EVENTS(LiveViewGLCanvas::OnMouseEvent)
END_EVENT_TABLE() //-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
LiveViewGLCanvas::LiveViewGLCanvas( wxWindow* parent, 
    wxWindowId id,
    const wxPoint& pos, const wxSize& size, 
    long style, const wxString& name) : wxGLCanvas(parent, (wxGLCanvas*) NULL, id, pos, size, 
    style|wxFULL_REPAINT_ON_RESIZE, name ), 
    m_init(false),
    m_gllist(0),
    m_interlacer(0),
    //m_bFlip(false),
    m_bSFReqToInit(true) {
```
m_bSeefrontResAvailable(false),
m_bSeefrontModeCheck(false),
m_bSeefrontAvailable(false),
m_iRenderMode(eSXS), // set default rendering to SeeFront
m_init_mil_file(true),
stop(false),
ipl_image1(0),
ipl_image2(0),
ipl_image3(0),
ipl_image4(0),
ipl_image5(0),
ipl_image6(0)
{
    // no impl
}

LiveViewGLCanvas::LiveViewGLCanvas( wxWindow * parent, const LiveViewGLCanvas* other, wxWindowID id, const wxPoint& pos,
const wxSize& size, long style,
const wxString& name ) :
wxGLCanvas(parent, other->GetContext(), id, pos, size, style|wxFULL_REPAINT_ON_RESIZE, name),
m_init(false),
m_gllist(other->m_gllist), //share display list
m_interlacer(0),
//m_bFlip(false),
m_bSFReqToInit(true),
m_bSeefrontResAvailable(false),
m_bSeefrontModeCheck(false),
m_bSeefrontAvailable(false), // set default rendering to SeeFront
m_init_mil_file(true),
stop(false),
ipl_image1(0),
ipl_image2(0),
ipl_image3(0),
ipl_image4(0),
ipl_image5(0),
ipl_image6(0)
{
}

// Destructor
LiveViewGLCanvas::~LiveViewGLCanvas()
{  
    // destroy the SeeFront interlacer object if it exists  
    if(1==sfogl::checkInstance(m_interlacer)){  
        // stop interacting with the tracker  
        sfogl::stopTrackerUpdate(m_interlacer);  
        // die  
        sfogl::destroy(m_interlacer);  
        m_interlacer = 0;  
    }  
    else{  
        // do nothing  
    }  
}  

// do nothing  

// setup for the rendering  
wxPaintDC dc(this);  
bool setupOK = SetupRenderParams();  
counter2++;  
wxString str;  
int x;  
test_timer2->Start();  
switch(m_iRenderMode){  
    case eSXS:  
        RenderSxS( dc );  
        break;  
    case eSF:  
        if(setupOK){  
            RenderSF( dc );  
        }  
        break;  
    default:  
        RenderSxS( dc );  
        break;  
}  

// RenderMode  
if(counter2%15==0){  
    x = test_timer2->Time();  
    str = wxString::Format( wxT("Render Image: %d ms\n"), x );  
    write_file2->Write(str);  
}  

// do nothing  

void LiveViewGLCanvas::OnKeyPress( wxKeyEvent& event )  
{  
    //send event to parent  
    event.ResumePropagation(1);  
    event.Skip();  
}  

// do nothing  

void LiveViewGLCanvas::OnMouseEvent(wxMouseEvent &event)  
{
{  
    // send event to parent
    event.ResumePropagation(1);
    event.Skip();
}

void LiveViewGLCanvas::OnEraseBackground(wxEraseEvent& WXUNUSED(event))
{
    // Do nothing, to avoid flashing.
}

void LiveViewGLCanvas::OnSize(wxSizeEvent& event)
{
    // this is also necessary to update the context on some platforms
    wxGLCanvas::OnSize(event);

    // set GL viewport (not called by wxGLCanvas::OnSize on all platforms...)
    int w, h;
    GetClientSize(&w, &h);
    SetCurrent();
    glViewport(0, 0, (GLint) w, (GLint) h);

    // resize the interlaced window
    if(eSF == m_iRenderMode && 1==sfogl::checkInstance(m_interlacer)){
        sfogl::resize(m_interlacer, w, h);
    }
}

void LiveViewGLCanvas::RenderSF(wxDC &dc)
{
    if(m_bSeeFrontResAvailable){
        // set the current context for OpenGL
        SetCurrent();
        // check if OpenGL has been initialised
        checkGLInit();
        sfogl::setTextures(m_interlacer, m_Images, 2, 0.0f, 1.0f, 1.0f, 
                           0.0f);
        sfogl::setTextureSize(m_interlacer, 1600,1200);
        sfogl::setScreen(m_interlacer,0,0,1600,1200);
        sfogl::render(m_interlacer);
        SwapBuffers();
    }
    else{
        // do nothing
    }
}

void LiveViewGLCanvas::RenderSxS(wxDC &dc)
{
    // set the current context for OpenGL
    SetCurrent();
    // check if OpenGL has been initialised
    checkGLInit();
    SetClientSize(848,280);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
// left view
glBindTexture(GL_TEXTURE_2D, m_Images[0]);
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f( 0.0f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f( 0.0f, 0.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f( 0.5f, 0.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f( 0.5f, 1.0f, 1.0f);
    glEnd();
    // right view
    glBindTexture(GL_TEXTURE_2D, m_Images[1]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f( 0.5f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f( 0.5f, 0.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f( 1.0f, 0.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f( 1.0f, 1.0f, 1.0f);
    glEnd();
glFlush();
glDisable(GL_TEXTURE_2D);
SwapBuffers();

void LiveViewGLCanvas::InitGL()
{
    // set context
    SetCurrent();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 3.0f);
    glTranslated(0,0,-2);
    glMatrixMode(GL_MODELVIEW);
    /* clear color and depth buffers */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

void LiveViewGLCanvas::checkGLInit()
{
    if(!m_init){ //has not been init'ed
        InitGL();
        m_init = true;
    }
}
bool LiveViewGLCanvas::SetTextures(GLuint* textures)
{
    // Check if images need to be flipped left<->right
    if(1){
        m_Images[0] = textures[0];
        m_Images[1] = textures[1];
    }
    Refresh( false );
    Update();
    return true;
}

void LiveViewGLCanvas::trackerUpdate(double x, double y, double z)
{
    // received new tracker info, simply repaint
    Refresh(false);
    Update();
}

void LiveViewGLCanvas::SetRendering(LiveViewGLCanvas::RenderMode mode)
{
    m_iRenderMode = mode;
}

int LiveViewGLCanvas::GetRenderMode()
{
    return(m_iRenderMode);
}

bool LiveViewGLCanvas::SetupRenderParams()
{
    bool setupOK = false;
    if( m_init_mil_file )
    {
        m_init_mil_file = false;
    }
    switch(m_iRenderMode){
        case eSXS: // side by side display
            SetClientSize(848,280);
            setupOK = true;
            break;
        case eSF: // interlacing for the SeeFront display
            // check if an instance exists of the interlacer
            setupOK = CreateSFInterlacer();
            break;
        default:
            // ! \TODO
            break;
    }
    return setupOK;
}
void LiveViewGLCanvas::SetImgDims(int width, int height)
{
    m_imagewidth = width;
    m_imageheight = height;
}

void LiveViewGLCanvas::SetSeefrontRes(bool val)
{
    m_bSeefrontResAvailable = val;
}

bool LiveViewGLCanvas::IsSeeFrontModeAvailable()
{
    // if not already checked..
    if(!m_bSeeFrontModeCheck){
        // valid gl context is necessary for the interlacer
        checkGLInit();
        // an instance does not exist,
        // try to create an instance of the interlacer..
        sfogl::IlaceHandle test = sfogl::createInstance();
        // has the interlacer been setup successfully?
        if(1 == sfogl::checkInstance(test)){
            // destroy the interlacer
            sfogl::destroy(test);
            m_bSeefrontCanInit = true;
        }
    } else{
        m_bSeefrontCanInit = false;
    }
    m_bSeeFrontModeCheck = true;
    return(m_bSeefrontCanInit);
}

bool LiveViewGLCanvas::CreateSFInterlacer()
{
    // did we check if the seefront mode is supported?
    if(!m_bSeeFrontModeCheck){
        IsSeeFrontModeAvailable();
    }
    // Can create an interlacer IFF :
    // (a)  m_bSeefrontCanInit --> The interlacer can actually be created
    // (b)  !m_bSeefrontAvailable --> The interlacer doesn't already exist
    if(m_bSeefrontCanInit && !m_bSeefrontAvailable){
        // simply create an instance...
        m_interlacer = sfogl::createInstance();
        // set tracker updates to the interlacer
        sfogl::setTrackerCallback(m_interlacer, &g_tracker_update);
        sfogl::startTrackerUpdate(m_interlacer);
        m_bSeefrontAvailable = true;
    }
    return(m_bSeefrontAvailable);
}
/call grabImages and set them as textures

void LiveViewGLCanvas::getTextures() {
    // convert these ipl images to texture without devIL if u use opencv
    ipl_image3 = cvCreateImage( cvSize( ipl_image1->width, ipl_image2->height ),
                                IPL_DEPTH_8U, 3 );
    ipl_image4 = cvCreateImage( cvSize( ipl_image2->width, ipl_image2->height ),
                                IPL_DEPTH_8U, 3 );
    cvCvtColor( ipl_image1, ipl_image3, CV_GRAY2RGB );
    cvCvtColor( ipl_image2, ipl_image4, CV_GRAY2RGB );
    for( int i=0; i<2; i++) {
        /// generate and bind GL texture
        glGenTextures(1, &m_testImages[i]);
        glBindTexture(GL_TEXTURE_2D, m_testImages[i]);
        // Use linear interpolation for magnification filter
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // Use linear interpolation for minifying filter
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        // Pixel data alignment adjustment
        glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
        // Texture specification
        if( i == 0 ) {
            glTexImage2D( GL_TEXTURE_2D, 0,
                            GL_RGB,
                            GL_UNSIGNED_BYTE, ipl_image3->imageData );
            } else {
                glTexImage2D( GL_TEXTURE_2D, 0,
                              GL_RGB,
                              GL_UNSIGNED_BYTE, ipl_image4->imageData );
                }
        SetImgDims( ipl_image1->width, ipl_image1->height );
        // send textures to renderer
        SetTextures( m_testImages );
        // clear storage
        glDeleteTextures(1, &m_testImages[0]);
        glDeleteTextures(1, &m_testImages[1]);
    }
}

void LiveViewGLCanvas::paintNow() {

wxClientDC dc(this);
    bool setupOK = SetupRenderParams();
    switch(m_iRenderMode){
        case eSXS:
            RenderSxS( dc );
            break;
        case eSF:
            if(setupOK){
                RenderSF( dc );
            }
            break;
        default:
            RenderSxS( dc );
            break;
    }
} // RenderMode

// RenderMode

// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef __LIVEVIEWFRAME_H__
#define __LIVEVIEWFRAME_H__
#define ID_RENDER_SXS 10000     // Side-by-side rendering
#define ID_RENDER_SF 10001     // seefront 3d rendering
#include <wx/wx.h>
#include <wx/display.h>     // multiple monitor support
//******* DevIL ***** //
#include <IL/il.h>
#include <IL/ilu.h>
#include <IL/ilut.h>
//************************
// Live View information
#include "LiveView_GLCanvas.h"     // Interlacing canvas
class LiveViewFrame : public wxFrame
{
public:
    //------------------------------
    // Create the live view display frame
    /*
       Creates and returns a reference to an instance of
       the frame
    */
    static LiveViewFrame* Create(LiveViewFrame *parentFrame, bool isCloneWindow = false);
    //------------------------------
    // Destroy the frame
    /*
Destroys the created frame (cleanup and window destruction)
/*
void Stop();
------------------------------------------------------------------------
public:
IplImage *m_image1;
IplImage *m_image2;
bool m_RENDER;
bool m_READ;
//------------------------------------------------------------------------

//! Set current image information
/*
Assign the live view image information to OpenGL
Textures (which are handled by LiveViewGLCanvas to render to the screen)
*/
public:
// ****** EVENT HANDLERS ******* //
void KeyPressEventHandler(wxKeyEvent& event);
void MouseEventHandler(wxMouseEvent& event);
//------------------------------------------------------------------------

//! Rendering mode : Set to Side-By-Side
void SetRenderingSxS();
//------------------------------------------------------------------------

//! Rendering mode : Set to SeeFront 3D Interlacing
void SetRenderingSF();
//------------------------------------------------------------------------

//! If we need an exit from the menu (not used currently)
void OnExit(wxCommandEvent& event);
//------------------------------------------------------------------------

void startWhileLoop();
// SetData generates the textures and sets them to render
void SetData();
private:
void GetDisplayInfo();
//------------------------------------------------------------------------

LiveViewFrame(wxWindow* parent,
wxString& title,
wxPoint& pos,
wxSize& size,
long style = (wxDEFAULT_FRAME_STYLE | wxNO_BORDER) & ~/*wxCAPTION |*/ wxCLOSE_BOX | wxSYSTEM_MENU | wxRESIZE_BORDER);
//------------------------------------------------------------------------
private:
    //---------------------------------------------------------------
    LiveViewGLCanvas *m_glCanvas;
    //! The OpenGLCanvas to draw the live view image to
    //---------------------------------------------------------------
    ------------------------------------------------------------------
    wxPoint m_dragStartPosMouse;
    wxPoint m_dragStartPosWin;
    bool m_bMoving;
    //---------------------------------------------------------------
private:
    //---------------------------------------------------------------
    IUInt m_Image[2];  //!< Current raw/jpeg encoded) data buffers
    GLuint m_glImage[2]  //!< OpenGL Textures
    //---------------------------------------------------------------
    private:
    //---------------------------------------------------------------
    DECLARE_EVENT_TABLE()
    //---------------------------------------------------------------

};  // class LiveViewFrame
#endif  // __LIVEVIEWFRAME_H__

/**
 * \file LiveViewFrame.cpp
 * \author Bishal Neupane
 * \author Pooya Moazzeni Bikani
 * \author 2012 Blekinge Institute of Technology . All rights Reserved
 * (18 May 2012) Bishal and Pooya
 * - Created file.
 */
// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#include "LiveViewFrame.h"
BEGIN_EVENT_TABLE(LiveViewFrame, wxFrame)
    EVT_CHAR(LiveViewFrame::KeyPressEventHandler)
    EVT_MOUSE_EVENTS(LiveViewFrame::MouseEventHandler)
END_EVENT_TABLE()
//----------------------------------------
LiveViewFrame::LiveViewFrame(  wxWindow* parent,
const wxString& title,
const wxPoint & pos,
const wxSize & size,
long style)
wxFrame(parent, wxID_ANY, title, pos, size, style)
{
    // init the OpenGL canvas
    m_glCanvas = NULL;
    m_bMoving = false;
    // init DevIL - for image handling
    iluInit();
    ilutRenderer(ILUT_OPENGL);
}

// true is to force the frame to close
Close(true);

void LiveViewFrame::KeyPressEventHandler(wxKeyEvent& event)
{
    event.StopPropagation();
    int code = (int)event.GetKeyCode();
    // 's' or 'S'
    if(115 == code || 83 == code){
        SetRenderingSF();
    }
    // 'x' or 'X'
    if(120 == code || 88 == code){
        SetRenderingSxS();
    }
}

void LiveViewFrame::MouseEventHandler(wxMouseEvent& event)
{
    event.StopPropagation();
    // only move if it isn't SeeFront mode
    if(m_glCanvas->GetRenderMode() != LiveGLCanvas::eSF){
        // start moving
        if(event.LeftDown() && !m_bMoving){
            m_dragStartPosMouse = ClientToScreen(event.GetPosition());
            m_dragStartPosWin = GetPosition();
            m_bMoving = true;
        }
    }
    // stop moving
    if(event.LeftUp() && m_bMoving){
        m_bMoving = false;
        int x,y;
        x = y = 0;
    }
}
GetPosition(&x,&y);
}

// move with mouse drag
if(eventDragging() & m_bMoving){

    // move if dragging is continued beyond a tolerance of 5 pixels
    int tolerance = 5;
    wxPoint currPoint = ClientToScreen(event.GetPosition());
    int dx = currPoint.x - m_dragStartPosMouse.x;
    int dy = currPoint.y - m_dragStartPosMouse.y;
    // calculate end position
    Move(m_dragStartPosWin.x + dx, m_dragStartPosWin.y + dy);
}
}

void LiveViewFrame::Stop()
{
    m_glCanvas->setStop();
    Close(true);
}

void LiveViewFrame::SetRenderingSF()
{
    if(m_glCanvas->IsSeeFrontModeAvailable()){
        GetDisplayInfo();
        m_glCanvas->SetRendering(LiveViewGLCanvas::eSF);
    }
    else{
        SetRenderingSxS();
    }
}

void LiveViewFrame::SetRenderingSxS()
{
    m_glCanvas->SetRendering(LiveViewGLCanvas::eSXS);
    SetSize(wxDefaultCoord,wxDefaultCoord,848,280);
}

LiveViewFrame* LiveViewFrame::Create(LiveViewFrame *parentFrame, bool isCloneWindow)
{
    LiveViewFrame *frame = new LiveViewFrame(NULL,
                                           wxT("Live View Player"),
                                           wxDefaultPosition,
                                           wxDefaultSize);
frame->m_glCanvas = new LiveViewGLCanvas(frame, wxID_ANY, wxDefaultPosition, wxDefaultSize);
frame->Show(true);
frame->m_glCanvas->count_count = 0;

// Default rendering options
frame->SetSize(wxDefaultCoord, wxDefaultCoord, 848, 280);
frame->m_glCanvas->ipl_image3 = NULL;
frame->m_glCanvas->ipl_image4 = NULL;
frame->m_glCanvas->ipl_image5 = NULL;
frame->m_glCanvas->ipl_image6 = NULL;

// global data structures
frame->m_image1 = NULL;
frame->m_image2 = NULL;
frame->m_RENDER = false;
frame->m_READ = true;
frame->m_glCanvas->write_file2 = new wxFile("client3render.txt", wxFile::write);
frame->m_glCanvas->test_timer2 = new wxStopWatch;
frame->m_glCanvas->counter2 = 0;
return frame;
}

void LiveViewFrame::SetData()
{
    if ( m_RENDER && !m_READ )
    {
        for ( int i=0; i<2; i++)
        {
            glGenTextures(1, &m_glImage[i]);
            glBindTexture(GL_TEXTURE_2D, m_glImage[i]);
            // Use linear interpolation for magnification filter
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            // Use linear interpolation for minifying filter
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            // Pixel data alignment adjustment
            glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
            // Texture specification
            if ( i == 0 )
            {
                glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, m_image1->width, m_image1->height, 0, GL_RGB,
                             GL_UNSIGNED_BYTE, m_image1->imageData );
            }
            if ( i == 1 )
            {
                glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, m_image2->width, m_image2->height, 0, GL_RGB,
                             GL_UNSIGNED_BYTE, m_image2->imageData );
            }
        }
    }
}
void LiveViewFrame::GetDisplayInfo()
{
  // We want to check if the resolution required for the
  // seefront interlacing is supported 1600x1200
  // and if the current resolution is set to the required value
  // If neither of these are true, set mode to side-by-side
  // at the current window resolution
  int reqH = 1600; // required horizontal res.
  int reqV = 1200; // required vertical res.
  int posX = 0;
  int posY = 0;
  int sfDisplNum = 0; // Display number for the SeeFront
  // (default 0 : primary)
  bool isReqSize = false; // is the display at the required resolution?
  // how many monitors are connected
  const size_t nDisplays = wxDisplay::GetCount();
  // get properties of each display
  \TODO Really need to do this better!
  for ( size_t i = 0; i < nDisplays; i++){
    // display object
    wxDisplay display((unsigned int)i);
    // resolution of the display
    const wxRect r(display.GetGeometry());
    const wxRect rc(display.GetClientArea());
    // check width/height
    isReqSize = ( reqV==r.GetHeight() ) &&( reqH == r.GetWidth() );
    if(isReqSize){
      sfDisplNum = (unsigned int)i;
      posX = r.GetX();
      posY = r.GetY();
      break.
    }
  }
  // Required resolution is not matched
  if(!isReqSize){
    // Don't do anything, simply set render mode to sxs
//m_glCanvas->SetRendering(eX3D::LiveViewGLCanvas::eSXS);
m_glCanvas->SetSeefrontRes(false);

} else{
    m_glCanvas->SetSeefrontRes(true);

    // ugly hack... for some reason the seefront render doesn't give
    // a perfect picture at 0,0
    // so we move it a little to give a better picture
    // \TODO Needs to be fixed!!
    // add offset
    int xOffset = 8;
    int yOffset = 6;
    SetSize(posX + xOffset,posY - yOffset,reqH,reqV);
}

}