BEE09:03

Building of a Stereo Camera System

Thom Persson

This thesis is presented as part of Degree of
Bachelor of Science in Electrical Engineering

Blekinge Institute of Technology
August 2009

Blekinge Institute of Technology
School of Engineering
Department of Applied Signal Processing
Supervisor: Abhishek Ivaturi, eXperience3D Sweden AB
Examiner: Siamak Khatibi

Abstract

This project consists of a prototype of a stereo camera rig where you can mount two DSLR cameras, and a multithreaded software application, written in C++, that can move the cameras, change camera settings and take pictures. The resulting 3D-images can be viewed with a 2-view autostereoscopic display. Camera position is controlled by a step engine which is controlled by a PIC microcontroller. All communication with the PIC and the computer is made over USB. The camera shutters are synchronized so it is possible to take pictures of moving objects at a distance of 2.5 m or more. The results shows that there are several things to do before the prototype can be considered a product ready for the market, most of all the camera callback functionality.

Acknowledgments

I start by thanking Linus Nilsson for helping out with the construction and design of the rig and electronics. Without him this project would not have been as feasible as it was. *Nord-Lock AB*, where Linus works, has contributed with several parts of the hardware construction, among other things aluminum parts, screws, some electronics and tools, for example CNC milling machine and lathe, to make customized parts. Linus also helped me find other companies which have conduced with parts for the hardware constructions.
All Motion Technology AB conduced with the step engine and *Teknikprodukter AB* conduced with the belt transmission, and I thank them as well for making this thesis project feasible.

Company/Person	Contact/Information
All Motion Technology AB	http://www.allmotion.se/
Linus Nilsson	linus.nilsson@nord-lock.com
Nord-Lock AB, Linus Nilsson	www.nord-lock.com
Teknikprodukter AB	http://www.teknikprodukter.se/

Table of Contents

Figure Index

Abbreviations

API	Application Programming Interface
CNC	Computer Numerical Control
DSLR	Digital Single-Lens Reflex Camera
FIFO	First In First Out
GUI	Graphical User Interface
LCD	Liquid Crystal Display
LSB	Least Significant Byte
MSB	Most Significant Byte
SDK	Software Development Kit
STL	Standard Template Library
TRS	Tip Ring Sleeve
USB	Universal Serial Bus

1. Introduction

This thesis project is made for the company eXperience3D Sweden AB and the result will be a prototype of a potential product. We have a non-disclosure agreement and therefore I cannot be too specific when I describe the design and implementation. eXperience3D Sweden AB will be referred to by name or as “company” and the term “we” refers to me and the company if nothing else is stated. The project involves designing a stereo camera rig where two cameras are mounted, with a software application that can control the camera position on the rig and the on-camera settings. It is most crucial that the shutter release on both cameras are closed at the same time, otherwise a moving object in the scene can ruin the 3D-effect. Also the cameras need to be horizontally aligned, otherwise post-processing alignment needs to be added.

We assume that the pictures taken with the rig is aligned and therefore no image processing needs to be implemented in this project. To view the images on a autostereoscopic display external tools for making the required interlacing will be used. The live preview would require some image processing but this is not fully implemented in this project due to the extra time needed to create a workable solution. We also assume that the current user is familiar with taking photographs and therefore only one mode, with all camera properties changeable, is implemented instead of having different layouts and controls for professionals, amateurs and beginners.

Each camera is mounted on a wagon that runs along a guide and the movement is motorized and controllable from the software application. Furthermore the rotation of the cameras can be set manually by the user to get better results. We use DSLR cameras from Canon in this project and their API *EDSDK* for interaction with the camera. The resulting stereo pictures is displayed on a 2-view autostereoscopic display. It would take to much time designing both the hardware and software and therefore I chose to focus primary on the software part because I am more experienced in that field.

For the construction of the rig I got help from my friend Linus Nilsson who helped me to design and build the rig and electronics. He also wrote the PIC-application which controls the step engine driver and the shutter release. Communication between the PIC and the computer is made with the USB protocol.

The software is written entirely in C++ and will be tested on a PC running Windows XP, 32-bit. The code is written in such a way that it is easy to port the application to another platform, for example I use libraries for threads and GUI components that are cross-platform. I spent most time with the camera related part in order to get it to work. The application is multithreaded to achieve better performance.

In the background section I discuss about other ways to get the stereo pictures and why we chose to use two cameras on a rig and some of the problems the different approaches had. The requirements from eXperience3D Sweden AB are specified in a separate section, Requirements. In the rest of the chapters I have, where plausible, tried to separate the descriptions of software and hardware. In the Design part of this report I describe how the structure of the prototype is designed and why we decided to do it in that particular way. The Implementation section depicts how each part of the hardware and software was constructed to achieve the results. In Testing I define the different test conditions we put up to see how well the hardware and software works, and also different cases for taking pictures. Performance part takes up how the tests went. After that I draw some conclusions and specifies what needs to be done in the future to have a product that is ready for the market.

2. Background

eXperience3D Sweden AB main field is to make applications for autostereoscopic displays. To get a stereoscopic image there needs to be at least two different images of the same scene with some distance between them in order to get the depth perspective. To the left in Figure 1 there are three objects in the scene at different distances and a picture of the scene is taken with two cameras. To get the stereo effect the display uses either a parallax barrier or a lenticular lens to separate the images to each eye. When viewed on the autostereoscopic display the barrier or lens will make the objects appear in front of, or behind the display. The focus point in the scene is the zero-plane and is at the same level as the display. In this example the focus point is on object 2 making it positioned on the monitor, object 1 will be in front and object 3 will be behind, or inside, the display. The problem to solve is to take the two pictures for stereo-viewing. There are both single- and multi-user displays, I will primary work with a single-user display from SeeFront GmbH[1]. The single-user displays uses only two pictures while the multi-user user several images, usually seven or nine to get the stereo effect.

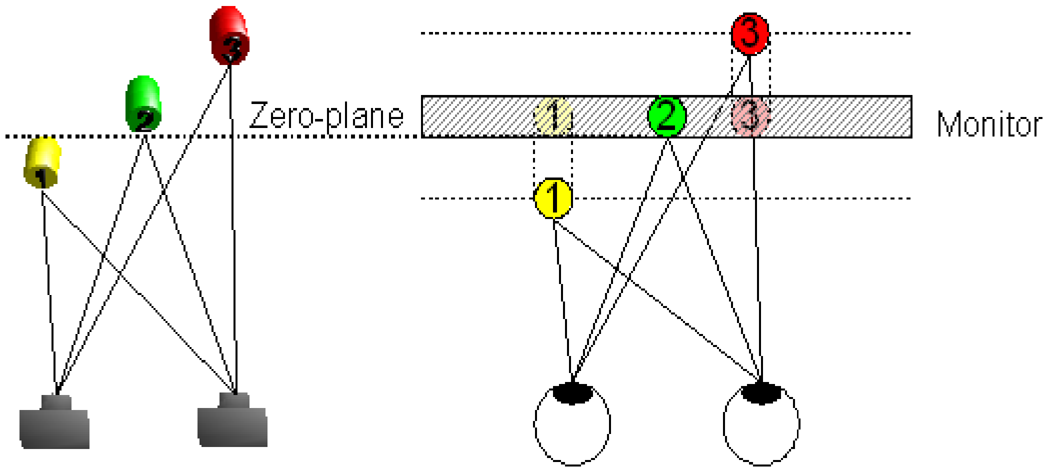


Figure 1: Picture taken with two cameras, viewed on a autostereoscopic display

There are several ways to make a stereo image pair. One way to do this is to have one camera and taking the two pictures by moving the camera slightly between each taken photo. A solution for taking this type of stereo picture is a sliding mounting device where you have, for example, a 10 cm long splint on a tripod with the camera attached to it and you slide the camera between the endpoints when you take the pictures. The problem with this method is that you cannot have moving objects in the scene because it takes time to slide the camera to the new position in order to take the second picture.

If you do not use the aforementioned sliding device on a tripod, but you take the pictures on freehand, there will be alignment problems because it is impossible to be at the same height with both shoots. Another way to do this is to have a one-piece stereo camera. This type of camera has two lenses, sensors etc., with a specified distance between them, base width, usually somewhere between 50-80 mm just as the distance between the human eyes. The problem with this technique is that you cannot change the base width or rotation and you do want to change the rotation and base width for several reasons, especially if you are a professional photographer.

1 More information available at http://www.seefront.com/

We have decided use a third way. It creates the stereo image pairs by having a rig with two cameras of the same model mounted on it. The pictures from the cameras are then interlaced in the computer to make a 3D-image. With the shutter release synchronized between the cameras you can get a good stereo picture even with moving objects. Because the cameras are mounted on a rail, or a similar device, rotation and distance control can be implemented on the construction.

To change the base width between the cameras means that it is possible to have distant object appearing closer on the resulting images, if you have the same ratio between the focus point (zero-plane) and the base width. With two cameras it can be hard to set the focus point at the exact same position. It is most critical that the focus point value are the same otherwise the resulting image can be ruined because, as mentioned before, the focus point affects where an object will be positioned in the 3D-image. If the cameras on the rig can be rotated inwards then it becomes easier to set the focus on the same object because you can rotate the cameras in such a way that the main object is in the center of both images.

3. Requirements

The result of this thesis project is a prototype of a potential product for eXperience3D Sweden AB. They have had no prior project that can be used together with this prototype. It will be both hardware and software development and it is possible to have the focus on one part. Trying to make both of them work well would take more time than a 10 week project. I have signed a non-disclosure agreement with eXperience3D Sweden AB and therefore I can only discuss the design and implementation parts on an abstract level, for example only showing pseudo-code instead of the exact implementation.

The end user is a photographer familiar with regular photography techniques, for example setting a good focus and white balance, but does not need to have any 3D-photography skills. This means that it is not necessary for him/her to think about synchronizing the shutter release, or the camera displacement. Those parts will be automated, or as a part of the software application where the user can change the functionality and/or get a readout of the current values.

To view the result a 2-view autostereoscopic display from SeeFront GmbH is used. Except the required interlacing to see the images on the display, made with Spatial View Inc. PowerPlayer[2], no additional image manipulation should be necessary. DSLR cameras will be used during implementation and testing, but any type of camera, both still and video, can be used on the rig in the future. If a live preview is available, i.e. the camera has a video output where you can review the scene before you take the picture, and the video stream can be transmitted to the computer, this should also be implemented.

The hardware part of this project is to construct the stereo camera rig. Two cameras will be mountable on it and their position and rotation can be changed by the user with good accuracy. The term "good accuracy" is highly subjective and depends on user preferences. For example one photographer changes the base width in centimeter steps while another changes it in a tenth of a millimeter. If possible the movements will be controllable from the software application. The rig should also be lightweight and as portable as possible to make it easier to move it around if the user takes picture outside of a studio environment.

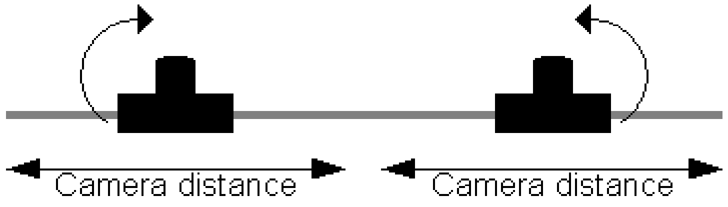


Figure 2: Conceptual camera movement and rotation on the rig

If an object moves in the scene it is crucial that the pictures is taken at the same time on both cameras, otherwise the object will be at a different position on the left and right picture and the 3D-effect of the resulting stereo image will be ruined. Therefore the shutter release must be synchronized and is one of the most crucial parts of this project. The cameras must also be perfectly aligned, otherwise the height offset will make artifacts on the resulting 3D-picture. It is possible to make a post-processing alignment, but as described above this is not a good solution because it requires additional manipulations of the image.

2 More information available at http://www.spatialview.com/

All code will be written in C++ and as portable as possible, meaning that it should be easy to build the software for Microsoft Windows, Mac OS X etc. For this project however the application will only be tested on a PC using Windows XP, 32-bit. Both the camera movements on the rig and the camera itself can be controlled from the software application, making the prototype completely remote controlled.

4. Design

The project is divided into three parts;

1. the rig where the cameras will be mounted, with camera position controlled from a step engine
2. a software application where the user can take pictures, change camera properties and position on the rig, and
3. the communication between them.

It would take to much time for one person to design all of this so for the construction of the rig, electronics and the majority of the communication between hardware and software we had a collaboration with my friend Linus Nilsson.

4.1. Choice of Camera

We decided to use Canon's DSLR cameras because the company had worked with the Canon API before and they already had one camera (EOS 350D) which could be used for initial tests. The model used later in the project is two EOS 450Ds which supports live view, i.e. a preview of the resulting image commonly featured on point-and-shoot cameras. The live view also gives us the advantage to change focus from the computer, which is currently not possible on Canon cameras with their API if the camera is not in live view, or does not support that feature.

A problem when working with Canons API is that it's very hard to get a synchronized shutter release. Instead of using the "take picture" command call through the API, I use the remote shutter port (a 2.5 mm TRS plug on the camera) to take the picture using the schematics from Covington (2004). We discovered that, when using only one opto-isolator, the potential difference between the cameras resulted in undefined behavior, i.e. the cameras took pictures at random. I made a modification on the circuit by adding an additional opto-isolator for the second camera, making the cameras completely isolated from each other. The shutter release are controlled from the PC as described below.

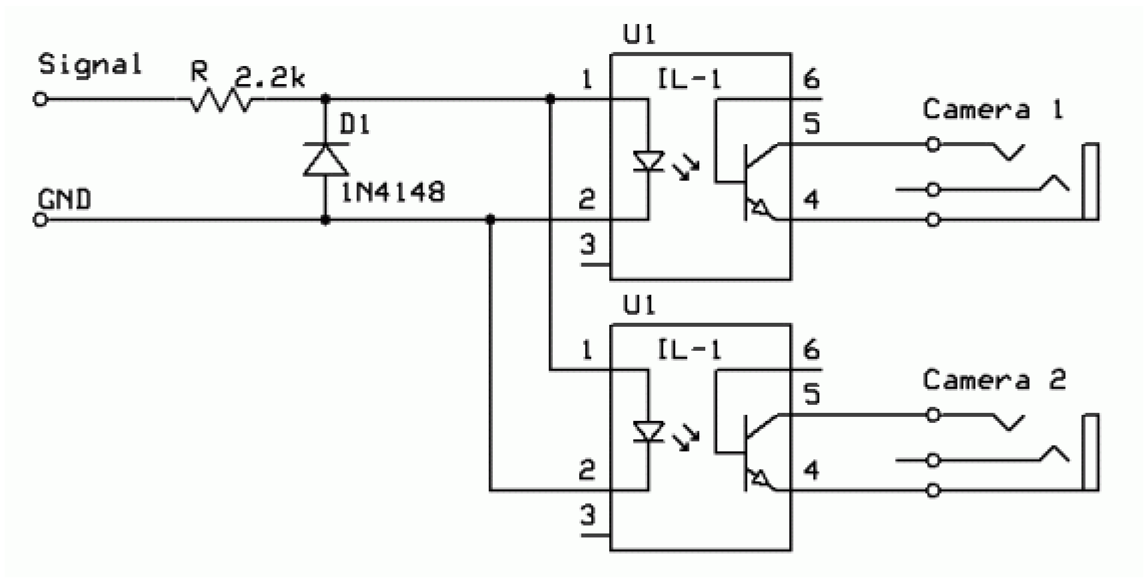


Figure 3: Schematics for the shutter release

4.2. The Rig

This section has been a collaboration between me, the company and Linus Nilsson. The term “we” will here be used to describe all three parts. To speed up the process of building the rig we decided to use existing wagons and a guide to mount the cameras on. Linus designed a special mounting device on the wagons for attaching the cameras. The mounting device also lets the user manually change camera rotation. Most of the parts are made of aluminum to keep the overall weight down.

Length of the factory-made guide is 596 mm which lets us use a wide base width. To change camera position, i.e. move the wagons across the guide, we use a step engine. The engine is attached to the wagons with a driving belt. Only one engine is used but the wagons are attached on opposite sides of the band pulley causing them to move equally much in opposite directions.

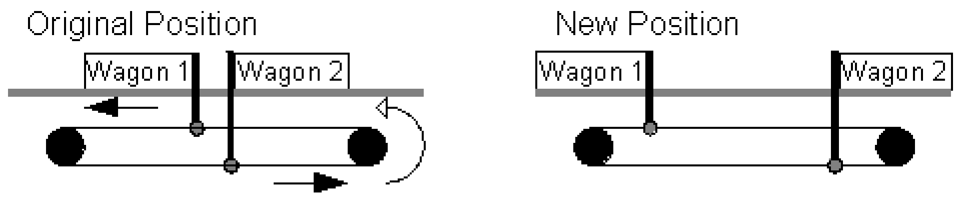


Figure 4: Engine rotating the belt counter-clockwise

Positions are displayed in mm to the user so it's necessary to convert the value to the corresponding size of a engine step. To do this there are some parameters to consider; lens-to-lens offset between cameras, band pulley diameter and steps per revolution in the engine. The formula is as follows:

p_{mm}	position, mm
p_s	position, engine steps
o_{cc}	camera offset, mm
s	steps per revolution
d	band pulley diameter

$$p_s = \frac{s(p_{mm} - o_{cc})}{d\pi}$$

A PIC[3] microcontroller controls the step engine driver and the shutter release. We also decided that the engine will be controlled from the computer over USB. To do this we used a FT232BM and the D2XX drivers[4]. Every instruction sent to the PIC microcontroller starts with a byte containing 10 (decimal value), followed by the command. If the new position value is beyond one of the endpoints on the rig, the value of the endpoint is returned. Conversion to and from engine step size is calculated before the value is transmitted to the PIC.

Table 1: Command layout

Command	Sent value	Returned value
Find Reference Point	'R'	'R'
Change Position	'G' MSB LSB	'G' MSB LSB
Shutter Release	'S'	'S'
Error	-	'E'

4.3. Software

Most time spent design and implement the application was devoted to get the camera related part to work. We use Canon's API called *EDSDK* to interact with the connected cameras. It is a C-style API, and some classes I designed works as wrappers to make the application structure more C++ like. It also uses asynchronous events, for example when a picture has been taken the camera sends an event when the image is ready to be downloaded to the computer.

The GUI is made with wxWidgets library[5]. We decided to use a native looking GUI with all controls accessible directly from the main frame. Other parts of the application, most notably the event handling also uses parts from wxWidgets. wxGlade 0.6.3[6] is used for designing the GUI. It lets you drag-and-drop wxWidgets components and generate code files based on the layout.

3 Programmable Intelligent Computer, microcontroller from Microchip Technology Inc. http://www.microchip.com/
4 Parts from Future Technology Devices Ltd. http://www.ftdichip.com/
5 Available at http://www.wxwidgets.org/
6 Available at http://wxglade.sourceforge.net/

4.3.1. Redesign

There has been two major redesigns of the software during the project. Those are related to the implementation of *EDSDK*. I knew, according to the EDSDK documentation (2008), that the *EDSDK* had problems with connections open to multiple cameras at the same time. After reading posts in the Canon SDK Group[7] I discovered that, if you are using threads, it would be possible to overcome this limitation. The first design was to have each camera object initialized in different threads. During the first month of development we only used one EOS 350D camera and the approach worked well.

When we switched to the newer model, EOS 450D, it was obvious that the implementation did not work properly. The only thing you could do with the camera was to start and stop the communication link. After trying different approaches the only solution to interact with the camera was to have the application in a single thread. Despite the documentation *EDSDK* did not have any problems with several connections open.

The new camera model also had support for live view (described in chapter 4.1). We decided to implement support for this. This decision lead to another problem. The live view output provided by the camera is not a streaming video, but an image object continuously updated on the camera. The only way to get a continuous, stream like, handling is to read this image object as soon it has been updated. The current design could not support this because it had to be done in the main thread, causing all other parts of the application, especially the graphical user interface, to get slowed down and be virtually unusable.

We made another redesign, now containing a command queue similar to the ones made by Eckel & Allison (2004). When implemented the camera callback functions did not work at all, but we decided to skip that feature and solve this problem outside of the thesis' scope. The final design consists of a user interface, camera interaction, hardware (rig controller) interaction and the communication between them. A simplified UML class diagram of the application can be seen in Figure 3.

7 http://tech.groups.yahoo.com/group/CanonSDK/

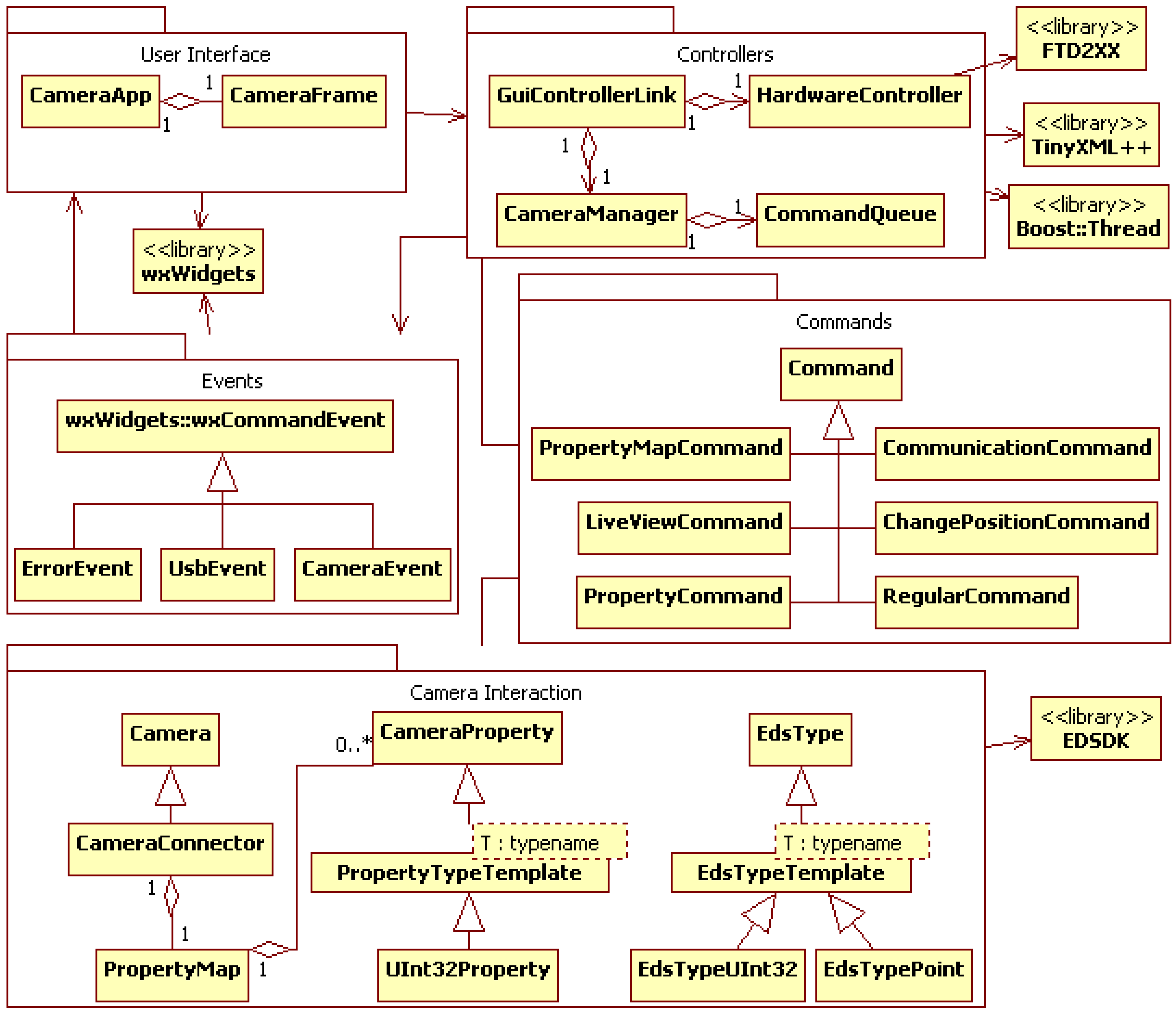


Figure 5: Diagram over the application

Table 2: Class description

Class Name	**Description**
Camera Interaction	
Camera	Wraps *EDSDK* functions and contains all the basic camera interactions, for example starting and stopping the communication.
CameraConnector	Inherits *Camera* and is extended for more interaction with the camera. Examples of additional functionality is; take picture, download picture and using live view.
PropertyMap	Stores *CameraProperty* objects together with a string value in a std::map. Some std::map methods are wrapped.
CameraProperty	A wrapper to bind a camera property from the *EDSDK* with a value.

Class Name	Description
PropertyTypeTemplate	A template to make interaction with *EdsType* derived classes easier.
UInt32Property	Wraps typedef integer from *EDSDK*
EdsType	A base class for wrapping typedefs in *EDSDK* into C++ classes.
EdsTypeTemplate	A template to make interaction with *EdsType* derived classes easier.
EdsTypeUInt32	Wraps typedef integer from *EDSDK*.
EdsTypePoint	Wraps a struct defined in *EDSDK* into a C++ class.
Commands	
Command	An abstract class with a execute method which will be called from *CameraManager*.
ChangePositionCommand	Change the placement identification on the camera.
CommunicationCommand	Start/stop communication with the camera.
LiveViewCommand	A live view related command.
PropertyCommand	Interact with a specific property.
PropertyMapCommand	Interact with all properties specified in a *PropertyMap*.
RegularCommand	Any other command.
Controllers	
GuiControllerLink	User inputs from *CameraFrame* are delegated to either camera or rig objects and events are sent back to CameraFrame.
HardwareController	All interaction with the *FTDI Chip D2XX* is made through this class.
CameraManager	Handles communication with cameras in a separate thread from the main thread. This lets the GUI continue to work while the camera handles the command.
CommandQueue	A thread safe queue where commands will be pushed and popped.
User Interface	
CameraApp	Implements *wxApp* from *wxWidgets* and is the entry point of the program.
CameraFrame	Implements *wxFrame* from *wxWidgets* and is the main frame of the application.
Events	
CameraEvent	Notification from a camera related class.
UsbEvent	Notification from *HardwareController*.
ErrorEvent	Notification from *GuiControllerLink* to *CameraFrame*. If this occurs user will be notified and an appropriate action is suggested.

4.3.2. Commands and Properties

One of the most crucial parts in the software consist of the aforementioned command queue and how a camera control event is processed. The commands are created and stored in *CommandQueue* from the main thread and then read from a working thread inside the *CameraManager* class. Because the commands are handled by multiple threads it has to be thread safe. I use thread classes from the *Boost C++ Library*[8] for this purpose.

The queue I implemented has a FIFO structure, so commands will be processed in the same sequence as they were pushed into the queue. If the camera is busy when a command is sent it will not be executed. In that case the command is put back into the queue for later processing. Also if the command for updating the live view stream it will put a new "live view update" command into the queue without notifying the user.

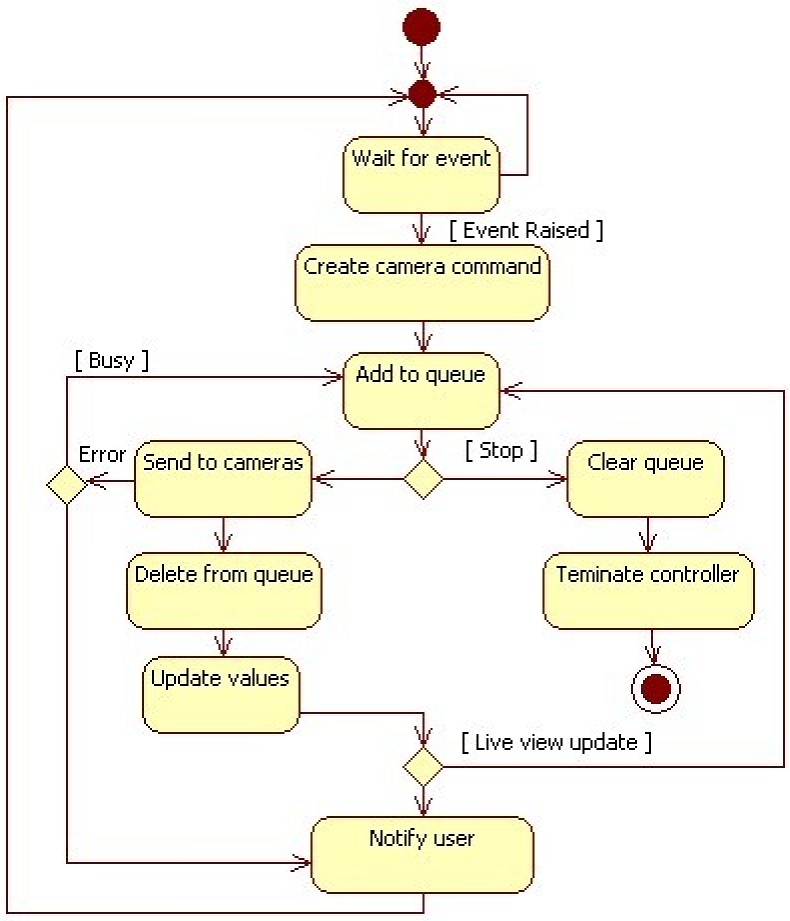


Figure 6: Handling a controller event

All camera properties, shutter speed, white balance, image quality etc., has its own definition in the *EDSDK* header files, using #define statements. The values for each property are either defined in the header files, or you need to look them up in the EDSDK documentation (2008).

8 Available at http://www.boost.org/

Instead of hard coding the defined values in maps or linked lists for easier look-up, we decided to store them in a XML data file. This way it is easier to add, remove or change a property and its values without having to recompile the application every time. To read the values into the program we use *TinyXML++*[9] which is small XML parser derived from *TinyXML*[10]. We also made a *XML Schema* based on the schema from *W3C*[11] using their data types. Part of the schema is included in Appendix 10.

9 Available at http://code.google.com/p/ticpp/
10 Available at http://www.grinninglizard.com/tinyxml/
11 Read more on http://www.w3schools.com/schema/default.asp

5. Hardware Implementation

All parts for the rig, except the guide and wagons, were made by Linus at his company Nord Lock using CNC based tools. He had almost full scope when designing the custom parts of the rig. Because of the traveling distance we did not have any on site meetings, instead we had meetings over the phone and email conversation.

The mounting device on the wagon is completely customized. It consist of two layers of aluminum plates which are attached to each other. The camera is attached to the device by unscrewing the bottom plate and tip it up. The bottom plate also has a path for the screw to move in, because the camera is fastened on the top plate, which can be rotated to adjust the inter-camera angle. The angle can be changed up to 45° per camera which gives a total rotation of 90°. The width of a wagon is 130 mm and when the wagons are at the closest distance, the reference point (described below), the lens-to-lens center distance is 149 mm. All screws has a hexagonal socket and most of them used to fasten the cameras, the plates and the rotation disc has the metric sizes 3 or 5. To mount the rig on the two tripods there are plates at the ends of the guide with regular ¼" with 20 threads per inch (TPI).

In the figure below we can see one camera attached to the wagon and the underside of the mounting device. The camera is rotated approximately two degrees and the wagons are at the reference point.

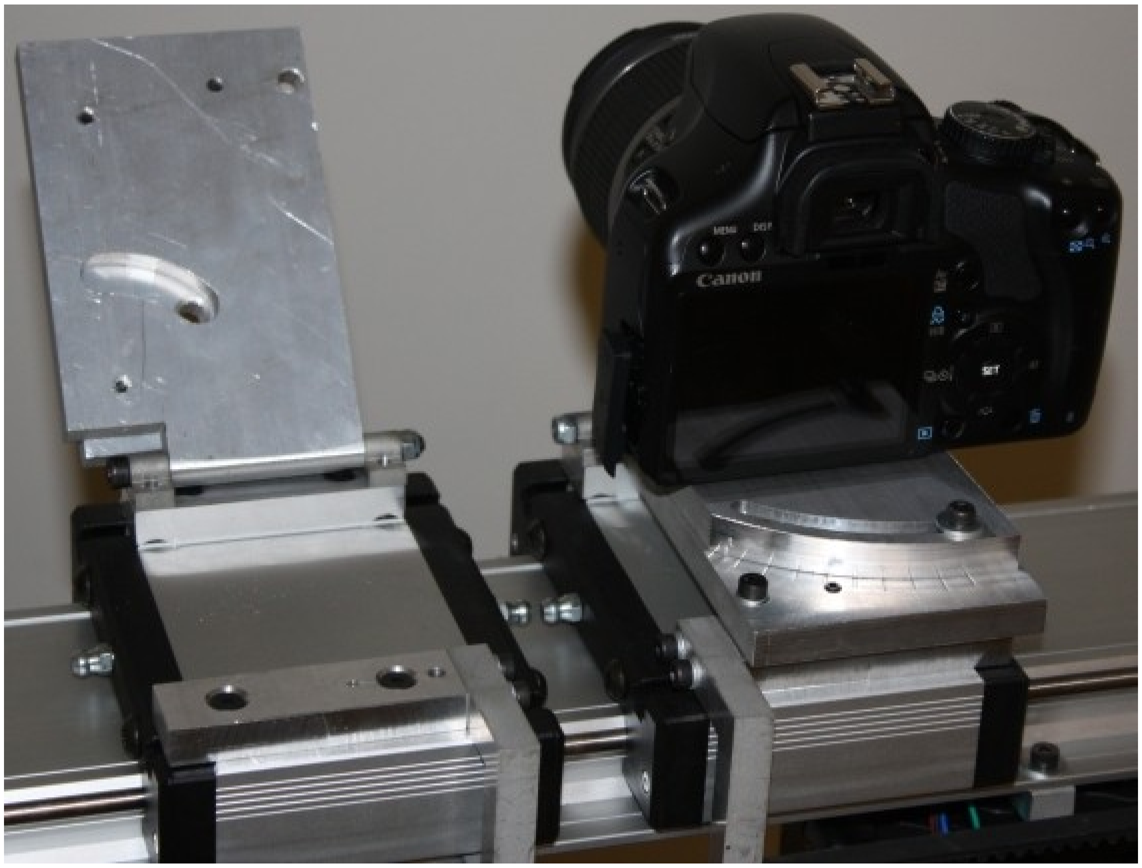


Figure 7: Camera mounting device

The driving belt, which moves the wagons, is attached in one end to an axle and the other end the step engine. With the wagons attached the maximum distance between the cameras, lens-to-lens center, is approximately 460 mm. This means that the whole length of the guide, which is 596 mm, is not used. 460 mm is still a good maximum value for the base width.

5.1. Electronics

The wagons are moved by a step engine located to the far right of the rig where it is attached to the driving belt. The engine is controlled with a step engine driver which receives instructions from the PIC. Linus made the program for the PIC. At the end points of the right wagon there are microswitches. The one at the center of the guide is the reference point, i.e. defined as zero in the PIC and the other one makes sure that the wagons stop at the edge of the guide. When starting up the controller it needs to be reset, i.e. the reference position must be reached before the wagon position can be changed.

We did talk about saving the last position in memory after each session. But if the rig's power supply is turned of there is no guarantee that the wagons will stay in the exact same position. This is because the wagons can easily be moved by hand when the engine is not holding them in plats. So if the wagons were moved just a little in between sessions the stored position value will be incorrect.

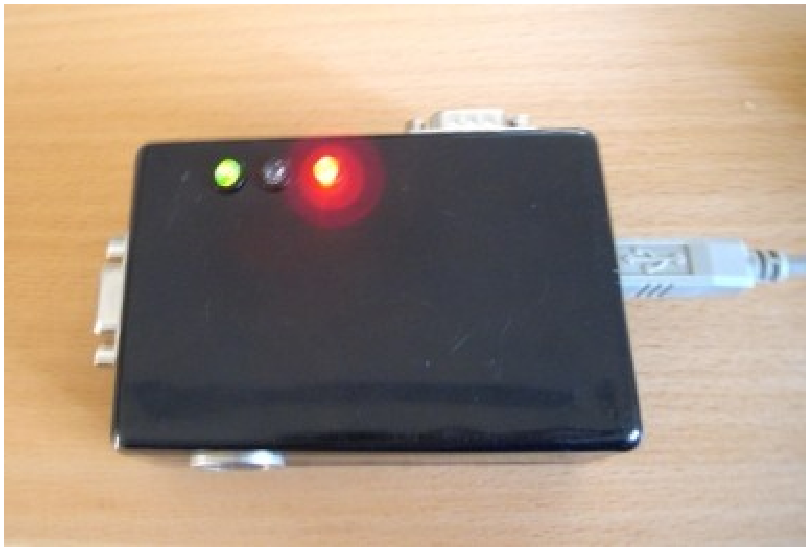


Figure 8: Taking Picture (note: no additional cables attached)

The part that I added to the circuit board was the shutter release described in chapter 4.1. To connect to the cameras we first used 2.5 mm TRS connectors in both ends of the cable. It turned out that they easily broke and after two setups using 2.5 mm connectors we switched to use 3.5 mm cables instead. The connectors on the controller box was changed and we used a 3.5-to-2.5 mm adapter for connecting to the camera. We also had problems with the connectors at the camera end. They were too long which led to that the right camera's wagon were prevented by the connector to reach the reference point microswitch. To solve this problem we used adapters with angled connectors.

6. Software Implementation

The project consist of a total of 34 classes. 26 of them interacts with *EDSDK*, thus most time has been spent on making these classes. Another factor for putting most of the time on the camera related code was that I needed to make the two redesigns mentioned in chapter 4.3.1. I have used a lot of containers from STL, especially vectors and maps in order to ease memory handling and iterations. Smart pointers are used as well, regular pointers are only used together with STL containers because std::auto_ptr does not work well in the containers, as described on Stack Overflow (2008).

EDSDK is C based and therefore uses void pointers for simulating polymorphism, typedef of struct for classes etc. Some of the methods in the *Camera* and *CameraConnector* classes wraps *EDSDK* functions that uses void pointers to get interaction from other parts of the application more C++ based. They also throws exceptions instead of returning the status value (defined in *EDSDK* header files). Exceptions are described in chapter 6.6.

EdsType and *CameraProperty* are base classes for all camera related data types. Each derived class contains of one data type defined in *EDSDK*. This is especially good when using containers from STL where you can store different data types using polymorphic pointers instead of having one container for each type. As seen in the diagram in chapter 4.3.1 there are two template classes, *EdsTypeTemplate* and *PropertyTypeTemplate,* derived from *EdsType* and *CameraProperty* respectively because a STL container needs to know the data type at compile-time. With the template classes it is easier to expand the data type support and still having different data types in one container element.

6.1. Threading

How the application uses threads has changed during the project together with the redesigns. The common part is that the *Boost::Thread* library has been used for all thread related interaction. *HardwareController*, not related to the camera communication, also uses threads but this use is described in chapter 6.7

Camera objects are stored in a vector, and for the first design each camera object was initialized in a separate thread. The threads were joined before the execution continued. For the 350D camera this worked well but when we switched to the 450D model nothing worked as before. The cameras did not respond to any commands sent to it. After testing to initialize the camera objects in the main thread it did work so we skipped the threads completely at this time.

6.1.1. Live View

As described in the chapter 3 we wanted to use the live view feature on the new camera model. The way that live view was implemented gave us another problem. First we thought that it was a video stream that could be handled by a regular stream handler, but it turned out that live view creates a single image object on the camera. The software is responsible for reading that data and to generate a continuous running video inside a picture box, or a similar graphical element in the application. To continuously read, convert and paint this live view image data, handling events, updating GUI elements etc. in the same, main, thread is not possible without getting a really long response time from the GUI. This is not a good solution and the user experience of the application would be really bad. We had to make a new multithreaded implementation.

The live view image is not displayed in the current application, only the metadata is processed. This contains information about where the zoom box is, current zoom and the focus. To view the current live view image it is necessary to look on the display on the back of the camera. The focus is somewhat interesting when using live view. If the camera is in live view and the focus is set to “Auto Focus” the regular half-pressed-shutter-auto-focus function does not work. Instead it lets you set the focus remotely from a computer using the auto focus servo engine. If you want to set focus directly on the camera it is better to use the “Manual Focus” mode when using live view.

6.2. Commands

All interaction with the cameras had to be in a thread, separated from the main (GUI) thread in order to get a better work flow. Also all calls to a camera was made directly from the main GUI classes to the C*amera* objects. With the continuous update of the live view images a potential problem with sending multiple calls to the camera at the same time had to be avoided. Live view is updated in the background without any directly visible notification to the user. If the user, for example, takes a picture and the live view data is updated at the same time two calls are sent to the camera without knowing that another part of the application is already accessing the camera. This can lead to a possible undefined behavior of the application. To overcome this problem I created the classes *Command, CommandQueue* and *CameraManager.*

Several classes inherits the *Command* class and one object of *Command* or a derived class contains one specific instruction to send to the *Camera* objects. For example a *CommunicationCommand* object starts/stops the communication link between the computer and the camera, and a *PropertyCommand* object reads or writes a property value on the camera. All these classes has a method called *execute()* which sends the stored command to the camera. All communication between *Camera* objects and the user are now made through *Command* objects.

6.2.1. Queuing

The class *CommandQueue* uses a std::queue where the commands are stored. It uses elements from the thread-safe multiple producer, multiple consumer queue class by Williams (2008). The class has basically two public methods, *push()* and *pop()*. To make it thread safe mutex locks and condition variables from the *Boost C++ Library* is used. If the queue is empty the consumer threads are put in wait mode. When a producer thread has pushed something into the queue it notifies one consumer which can pop the first command in the queue.

When a command is sent to the camera and you get a “Device Busy” or “Object Not Ready” response it is necessary to try to send the command again. This needs to be delayed for some time before it is processed again in order to let the camera get out of the busy state. A catch statement for camera related errors puts the thread into sleep for 500 ms and then puts the command back into the queue for processing again. This functionality is used extensively when live view is running because it takes time for the camera to buffer up the preview images.

CameraManager is initialized in a thread and has a main loop which is responsible for executing commands that is in the *CommandQueue* object. When the *CameraManager* object is about to be destroyed a *CommunicationCommand* with a “Stop Communication” instruction is executed and all other commands are removed from the queue. The camera objects are initialized in this thread as well making them separated from the main thread.

6.3. Problem With Camera Callbacks

Yet another problem occurred with the new design. The callback functions from the cameras did not work when the camera objects were in another thread. I tried to initialize *EDSDK* in several places, if initialized in the main thread it found the connected cameras but were not able to interact with them, i.e. when trying to establish a communication link with a camera you only got a “Device not found” response. If *EDSDK* is initialized in the other thread, in *CameraManager*, all functionality except the callbacks from the camera worked. This means that we cannot automatically download a picture from the camera directly after it has been taken and we do not get notified if, for example, the user changes the shutter speed on the camera. Because of the time already elapsed we decided to leave out the callback functionality for now, but we still wanted the application to automatically download the image.

To create a automatic download functionality I first tried to add the download instruction directly after a picture had been taken. It sometimes takes longer time for the camera to create the image object and the previous image is downloaded instead of the last one. The local image object counter on the camera did not always update, i.e. when a new session started only the first image object were recorded in the counter. I tried to force the application to read another non-recorded image object by having a local counter in the application, but it did not work. The only response I got was a “Object not Found” status message. The next thing I tried was to stop the current communication link, start it again and then download the last image. This approach worked well so we decided to use it.

One problem with the start/stop approach is when you are using live view. Because you have to look at the back of the camera to see the current image, by manually setting the LCD to show the live view, this will be turned of when the communication link is stopped and it is not possible to remotely set the camera to display the live view on the LCD.

6.4. User Interface

The graphical user interface is made with the *wxWidgets* library and the layout is made by using *wxGlade*. The layout is pretty simple with all related controllers, for example properties and live view, grouped together. The property combo boxes are filled with values stored in the property XML file, described below.

When the user clicks the start button a progress dialog appears to inform the user that something is happening. First the camera communication links are established and then the on-camera properties are read. If they differ they will be synchronized with the settings from the first connected camera. The only property that does not get synchronized is the *Shooting Mode*, for example “portrait”, “full auto” and “manual”. The reason is that it is set on the camera by turning a dial and you cannot set this remotely from the computer. If *Shooting Mode* differs then a pop-up notifies the user and that she/he needs to change this manually. If everything was successful the USB communication with the rig controller starts, else a pop-up describes the problem and does not continue the initialization.

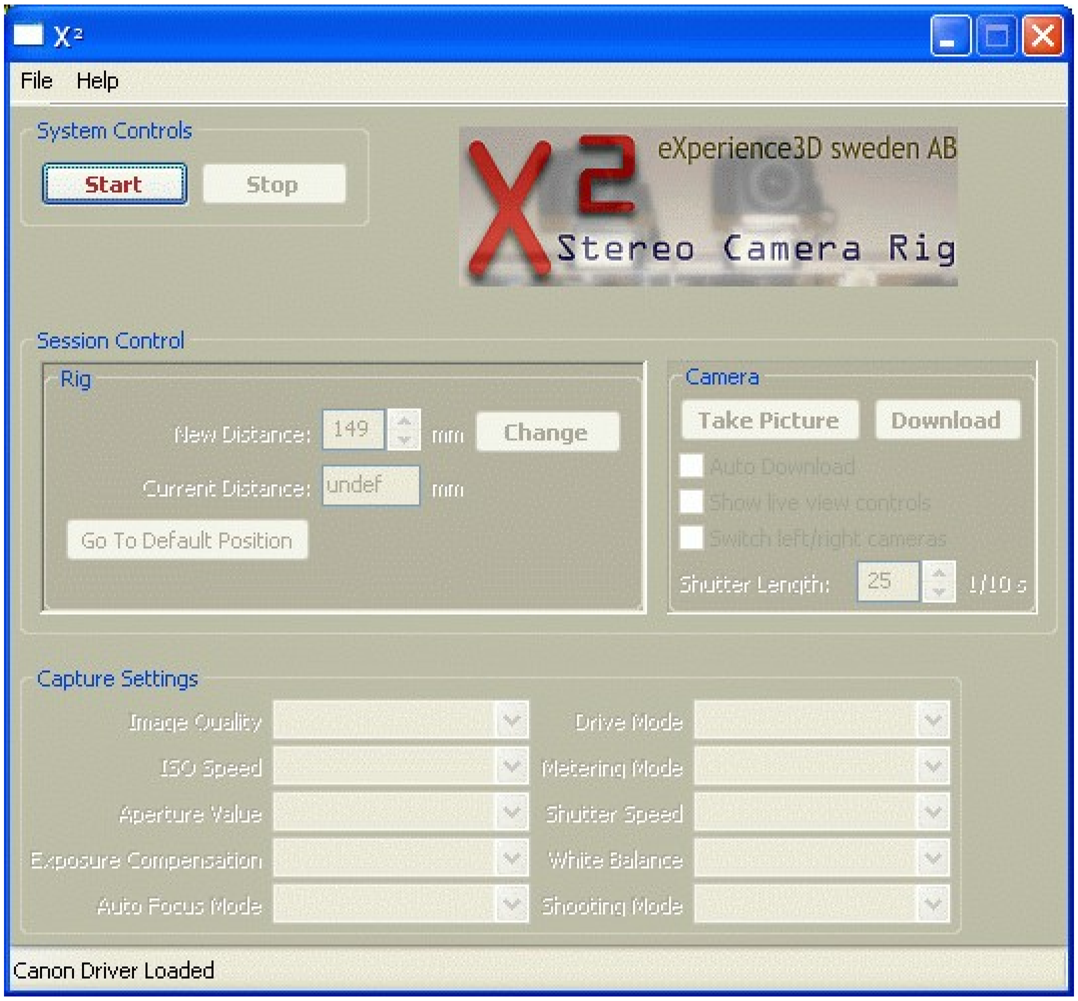


Figure 9: Application directly after start-up.

6.4.1. Events

To communicate between the GUI classes and the controller classes, *CameraManager* and *HardwareController*, *GuiControllerLink* is used. The GUI classes receives events raised from the user, for example changing a property setting, and calls the corresponding method in *GuiControllerLink* which sends the instruction to either camera or rig controller depending on which event was raised. The controller object executes the instruction and when it is done an event is sent back to *GuiControllerLink*. To communicate with events between the controller classes is better because of the detached threads, as described in chapter 6.6.

The events sent from the controller classes are derived from *wxCommandEvent* which is one of the event classes in *wxWidgets*. This way I did not have to write any event handlers, I only had to implement the event handler from *wxWidgets* in *GuiControllerLink*. What the custom events all have in common is that they have a string which describes the event and the executed instruction.

6.5. Storing Data in XML File

Property information and values are stored in a XML file which makes it easier to change without having to recompile the application. The *TinyXML++* parser, built on top of *TinyXML,* is easy to work with because it uses iterators to go through nodes in a XML document and exceptions if something goes wrong.

Data is read from the file when the application starts up. If the file is missing or corrupt the user gets notified about this and is informed to take appropriate actions. The data is stored in a struct which is loaded into the property combo boxes in the GUI's lower area.

6.6. Error Handling

Most part of the error handling is made by throwing exceptions. I made a special class for this and if an error occurs an object of this, or a derived class, is thrown. The derived classes are all specified to handle a specific group of errors, for example camera (*EDSDK*) exceptions. The classes also contains information about where the exception occurred and a human readable string describing the problem.

To use exceptions makes the code generally more readable. When you look at examples from EDSDK documentation (2008) you see that there is a lot of nestled if-statements. Instead when using exceptions you just have to use an if-statement to see if the previous operation were successful, and if not throw an exception. You do not have to have return values from each method either, instead you execute the methods inside a try-catch block and lets the catch block handle the exception or re-throw it. Consider the following code example using Canons way of coding:

```
EdsError initializeEDSDK();
EdsError connectToCamera();
EdsError readProperties();

EdsError start()
{
      EdsError err = OK;

      err = initializeEDSDK();

      if (OK == err){
            err = connectToCamera();

            if (OK == err){
                  err = readProperties();

                  if (OK == err){
                        // Do something
else...
                  }
            }
      }

      return err;
}
```

Code 1: if-statement nested code
There is a lot of if-statements and for each new method call another block is needed. The calling method also needs to handle the return value.

```
void initializeEDSDK();
void connectToCamera();
void readProperties();

void start()
{
      EdsError err = OK;

      try{
            initializeEDSDK();
            connectToCamera();
            readProperties();
            //Do something else...
      }
      catch(EdsError e){
            throw e;
      }
}
```

Code 2: try-catch based code

If we compare the last code example with the previous one we see that the readability is simplified. The three methods are changed to void instead of returning the *EdsError* value, which will be thrown from the method instead. If we use the exception class here instead of throwing the *EdsError* we also get information about where the error occurred and it is easier to debug or give the user a better description of what went wrong.

The application is multithreaded using detached threads and it is not a good idea to have exceptions thrown out of them. It is analogous with throwing something out of the *main()* function, as described by Crocker (2008). Almost all exceptions are re-thrown up to *CameraManager* or *HardwareController* which has their own top-level exception handlers. These classes converts the exceptions into events which are handled by *GuiControllerLink*.

6.7. Hardware Interaction

The *D2XX* driver API has two different interfaces, the “classic” and a *FT-Win32* version. The latter one uses function calls similar to the *Win32 COMM* API. I choose to use the *FT-Win32* version because, as the D2XX Programmer's Guide (2006) describes, is more sophisticated and if we want to change to another Win32 based driver we basically just have to change the function call names.

When *HardwareController* sends an instruction to the PIC microcontroller it is processed in a separate thread. The reason for using a thread here is that it takes some time to send the instruction and receive a response. For sending an instruction the timeout is one second before the operation is canceled and a exception is thrown to *GuiControllerLink*. The timeout when waiting for a response is on the other hand 20 seconds because it takes some time to move the wagons from the end positions. I also added some extra seconds to be sure that it actually moved all the way. To have this in the main thread would make the application less responsive and therefore it is running in a separate thread.

6.7.1. Commands

In chapter 4.2 I described the layout of the instructions sent to the PIC processor. Here is a more detailed description of them:

Find Reference Point locates the reference point on the rig. This is done by moving the cameras towards the center of the rig until they encounter a microswitch. This command must be sent before any attempt to change camera position is made, because the PIC microcontroller does not know where the wagons are.

Positioning tells the PIC to move the camera wagons to a new position relative to the reference point. If the reference point has not been located it returns error ('E'). Position value is sent from the PC in a two-byte size format, i.e. a conversion from integer value representing mm must be done before the command is sent, see Byte Buffers for description.

Take Picture takes a picture with the cameras.

6.7.2. Byte Buffers

The D2XX driver write/read byte size buffers to and from the hardware device. Therefore it is necessary to convert any values into these buffers. We decided to use a maximum of two bytes for specifying a position on the rig. This gives the maximum distance 65535 mm which is higher than the length of the rig. The complete command is stored in a std::vector containing unsigned chars. The data type char has the native size of a byte.

A “Positioning” command has the following structure:

Table 3: Positioning command

Vector element	**Value**
0	10
1	'G'
2	MSB
3	LSB

The distance value is stored in the *HardwareController* object as an unsigned integer. To convert the value into two chars we need to get the twp first lowest bytes. By using two filter masks, defined in hexadecimal as 0x00FF (LSB) and 0xFF00 (MSB), and applying an AND operation with the masks on the value we get the bytes. The MSB value must be shifted down one byte in order to store it in a char. All data types that interacts with the *D2XX* driver is unsigned because in the conversions the sign bit can make the conversion give a wrong value.

7. Testing

We took several groups of pictures to test the complete setup. By looking at the resulting 3D-picture on a SeeFront display we could directly see the current test cases. We tested to take pictures of the same scene and changing the base width. For example having a object three meters away from the rig and taking pictures of this with a increment of the base width by five millimeters.

Rotation was tested by changing the value when the cameras are at the closest point (reference point). The test object was moved back and forth in the scene to see where the object will be to close, i.e. you have to strain your eyes really hard to see the object. To test different focal points we made test groups where we had one object at a specified position. Then we changed the focal point to be in front of the object, on the object, behind the object etc. and compared the results.

Mixes of all of these tests were also made, where we changed the base width having different camera rotation, and changing focal point with different rotation.

7.1. Software

The software was tested during the development. We focused hard on making it as stable as possible and all messages that is displayed to the user is easy to understand, at least he/she should know from which part of the application the error occurred. For example if the camera failed to execute a command, which command and why. The different design stages were tested thoroughly before we switched to a new design in order to know which part of the software that needed to be redesigned. Response times between the user and the application were also tested and if it took too long time we tried to optimize that part of the code.

7.2. Hardware

Linus did the first tests of the rig and the electronics. This included to see if the engine driven wagon movement worked and that the PIC program worked. He made a test application where instructions could be sent to the PIC from the computer. It did not convert any values from the byte stream, just showing the last response value in raw numbers. I used this program for some time as well during testing of the rig and for comparison of the functionality with my own application. To test the movement and positioning we changed position from the application and then measured it with a ruler. Of course a ruler is not the best instrument to use for precise position measurements. For the scope of this thesis, and the fact that most of my focus where on the software part we decided that using a ruler would give a good enough result.

After adding the shutter release components into the electronics made by Linus, we also tested this functionality. We compared this with using *EDSDK* to take a picture in three different ways. First of all you can hear the shutters going off and this gives a good subjective perspective. If the shutters are far apart you can easily hear this, but when they are close together it is harder to determine the exact time between them. The next test is to look at the resulting pictures taken on movable objects. This way you can see if the object has moved between the left and right picture. The third way we tested it is a modification of the second version. It is the most objective test where we took pictures of a stop watch and the resulting pictures shows the actual time when they were taken.

8. Performance

Because of the extra time it took to make the new designs of the software there were no time to do thorough test of all parts. Overall the software response time is good, the only problem is when live view is enabled. Then it takes about 1-2 seconds before you can see that something actually happened with the camera. The only time the software becomes unstable is when a camera gets disconnected or runs out of battery when a connection is open. Because there is no notification from the camera about this event *EDSDK* tries to read/write to a invalid place, i.e. using a pointer to a deallocated memory region, and the program crashes.

Because the rig is mounted on two tripods it takes some time to set it up. You first need to see that the tripods are stable enough to hold the rig, i.e. standing on solid ground. Then the height must be adjusted to prevent the rig from tilting. We use a spirit level to see that the rig is aligned. It also takes time to mount the camera and because the mounting plates are quite large you need to remove the camera in order to remove the battery for charging. For one person to set up the rig, the autostereoscopic display, connect to the computer and starting the application it takes approximately 10-15 min.

8.1. Image Result

The shutter synchronization is good. There is no problem with taking pictures of a moving object and still get a good stereo image. Of course the object should not move to fast because it will give ghosting on the images just as when you take a regular picture. The standard lens which comes with Canon EOS 450D works well to take stereo pictures with because it has a large depth of field, i.e. most of the objects still has focus when the focus point is set on a object. A 3D-image should have the complete scene in focus to give a good depth perspective. We tested to have shallow depth of field by taking pictures of close objects. For example focus on a twig at a distance of 150 mm which leaves the rest of the tree blurred out. The twig itself gave a good 3D-perspective as it comes out of the monitor but the blurred parts of the image was just as a texture on a wall behind the twig with no changes in depth at all. We found that, with the closest base width possible (149 mm) and the camera set in parallel, the object in focus can be at a closest distance of 2.5 m. With some rotation the object could get even closer, but the base width constraint made it harder for the eyes to see the images.

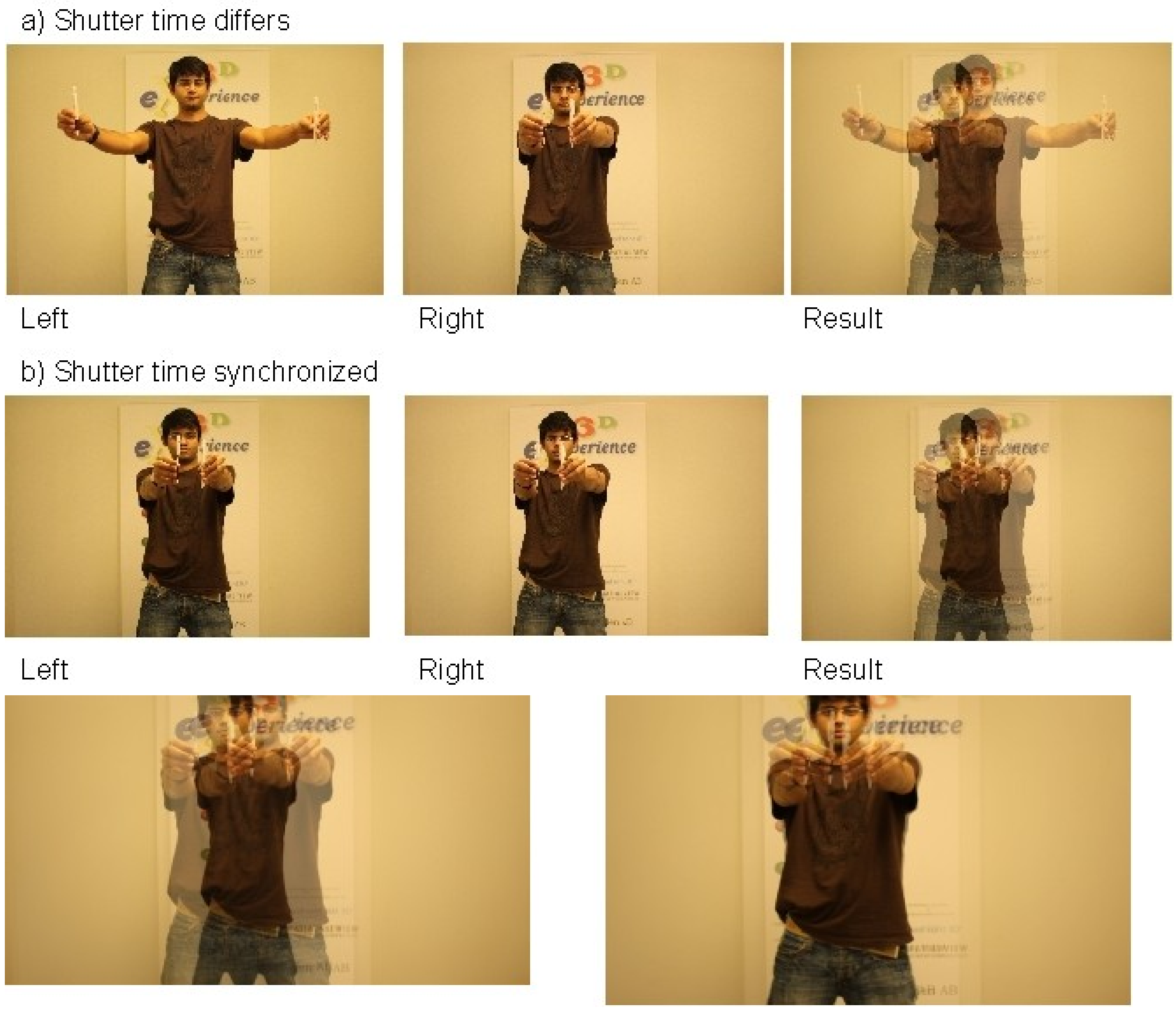


Figure 11: Alignment comparison

9. Results and Conclusions

When reviewing the resulting pictures on the autostereoscopic *SeeFront* display we get good results with the objects in the scene at a distance of 2.5 m or more. The default lens that comes with the EOS 450D camera is good enough to take stereo photographies with because of the large depth of field. The shutter synchronization is really good as well.

Positioning of the cameras works well and the communication using the USB module does not fail, unless you disconnect the cable. The decision to have two tripods to mount the rig on made the it less portable, but considering the weight and the current tripods we are using, the prototype needs both of them in order to keep the rig from falling.

But it is also clear that there has to be several adjustments on the product before it can be considered as a product ready for the market. The biggest problem with the software is the lack of callbacks from the camera because of the multithreaded environment. Some other way to separate the main (GUI) code from the CPU consuming update of the live view image objects must be implemented to fix this. The response time from the camera is very long when live view is running, it can take up to 1-2 seconds before you actually see that the camera has received the command.

With the rig there are some problems as well. Not only related to the rig, but also to the size of the DSLR:s, is the minimum camera displacement by 149 mm, which makes it impossible to use the human eye base width which is about 50-80 mm. The guide weights about 2 kg and the wagons 0.5 kg a piece. With the customized parts the overall weight is about 5 kg, excluding the cameras (about 0.5 kg each). The rig must be reduced by at least 2-3 kg to be considered lightweight.

The resulting pictures are misaligned because, even after adjusting the mounting plates, the right camera has two feet on the plate which gives an offset between the cameras. We can apply post-processing alignment on the pictures, but if we are going to use the live view the cameras must be aligned in order to get the video preview in real time.

If Canon's *EDSDK* would have been better implemented, for example natively supporting multithreaded applications, and the live view functionality would have been a simple video stream more time could have been spent on making the prototype better. For example fixing the live preview directly on the autostereoscopic display and making different modes like beginner, amateur and expert. More time could also have been spent on testing the rig. Now there was not time enough to make a thorough analysis of the stereo images.

10. Future Work

To have a product ready for the market there are several things to do. First of all the camera callbacks needs to be fixed to have a better interaction with the cameras. Without this there are several potential problems, for example the camera's runs out of battery and shuts down, which does not notify the application and can in a worst case make it crash. The current need to make workarounds to download an image directly after it has been taken does not work either in a product for the market. The application also needs to be tested on several different platforms, those supported by Canon's *EDSDK,* like Windows Vista and Mac OS X.

Because I did not have the time to implement the full support for live view this has to be done. In conjunction with this the alignment on the rig's mounting device has to be perfect, it is not a good idea to have live preview running and applying post-processing on the image stream. It will be really slow and virtually unusable because of the long processing time.

The hardware construction needs to be even more remote controllable. Now only the inter-camera displacement can be changed. The camera rotation should be motorized as well. To fix the alignment problem there must be a way to set the height of each camera with good accuracy, or a similar solution, must be implemented. The overall weight must be reduced as well. Changing material to a more lightweight material and not using parts from other manufacturers, but constructing it all from scratch to have a better overview of all parts. When the weight is reduced it will also be more portable. It is also a good idea to only use one tripod.

Appendix I - Camera Rig and Related USB Control Box

(The following text is part of a presentation given to me and eXperience3D Sweden AB by Linus Nilsson, written in Swedish, together with the rig and controller box.)

Kamerarigg och tillhörande USB-kontrollbox

Information

Denna utrustning är en prototyp som framtagits av Linus Nilsson som en del av Thom Perssons examensarbete vid Högskolan i Kalmar, för företaget eXperience3D's räkning. Riggen samt tillhörande USB-kontrollbox (inkl. ett testprogram) har konstruerats* och byggts av Linus Nilsson. Äganderätten till denna utrustning övergår vid överlämnande till Thom Persson, till eXperience3D. Upphovsrätten till kontrollboxens mjukvara förblir Linus Nilssons, om inget annat avtal sluts kring detta. Dock skickas den kompilerade mjukvaran med, om man vill bygga fler kontrollboxar.

** Thom har konstruerat och byggt kretsen för triggning av kamerornas slutare från en TTL-signal (aktiv hög).*

OBS!

Utrustningen är en prototyp, och ska ej betraktas som en färdig produkt. Den levererade strömförsörjningen är ett nätaggregat av industriell typ, och bör monteras i lämplig inkapsling på den slutgiltiga produkten. Det är upp till användaren att kontrollera att utrustningen är säker att anslutas till elnätet. Kontrollera t.ex att kabeln som går mellan vägguttaget och nätaggregatet sitter rejält fastskruvad i detta.

Funktion

Riggen består av två delar:

1. En linjärgejder med två st. vagnar som drivs av en stegmotor och en tandremstransmission. På vagnarna sitter varsitt fäste som har justerbar lutning och är vridbart 45 grader inåt.
2. En kontrollbox som kopplas till datorn via USB. Denna tar emot styrkommandon från datorn och styr motor samt kameraslutare.

Appendix II - Project Specifications

[Overview]

To generate a stereoscopic pair of photographs, a stereo-camera mount, allowing for remote camera synchronization is required.

The resulting pictures can be viewed in any autostereoscopic display using two-views.

The goals of this project are

(a) develop a model or model diagrams detailing a stereo camera rig including mount and camera synchronization, for still photography/videography

(b) implement a working prototype based on (a)

The project will include development of software required for remote interaction and control and hardware/software required for

synchronization of the stereo cameras

[Schedule]

The project is planned for a time period of 10 weeks, suitable for a 'C'-level Thesis Project.

An initial plan is proposed as follows. A detailed plan for each stage will be discussed after the student has been accepted

(1) Introduction, Project Definition	-	1	Week
(2) Literature/Preliminary Study	-	1	Week
(3) Design and Prototyping	-	2.5	Weeks
(4) Software Design, Implementation	-	3.5	Weeks
(5) Testing, Experimentation, Validation	-	1	Week
(6) Documentation	-	1	Week

[Prerequisite]

- Candidate for Degree or Diploma in Computer Science/Electrical Engineering or related
- Prior Programming Experience (C++ is required, MATLAB/OpenGL are a plus)

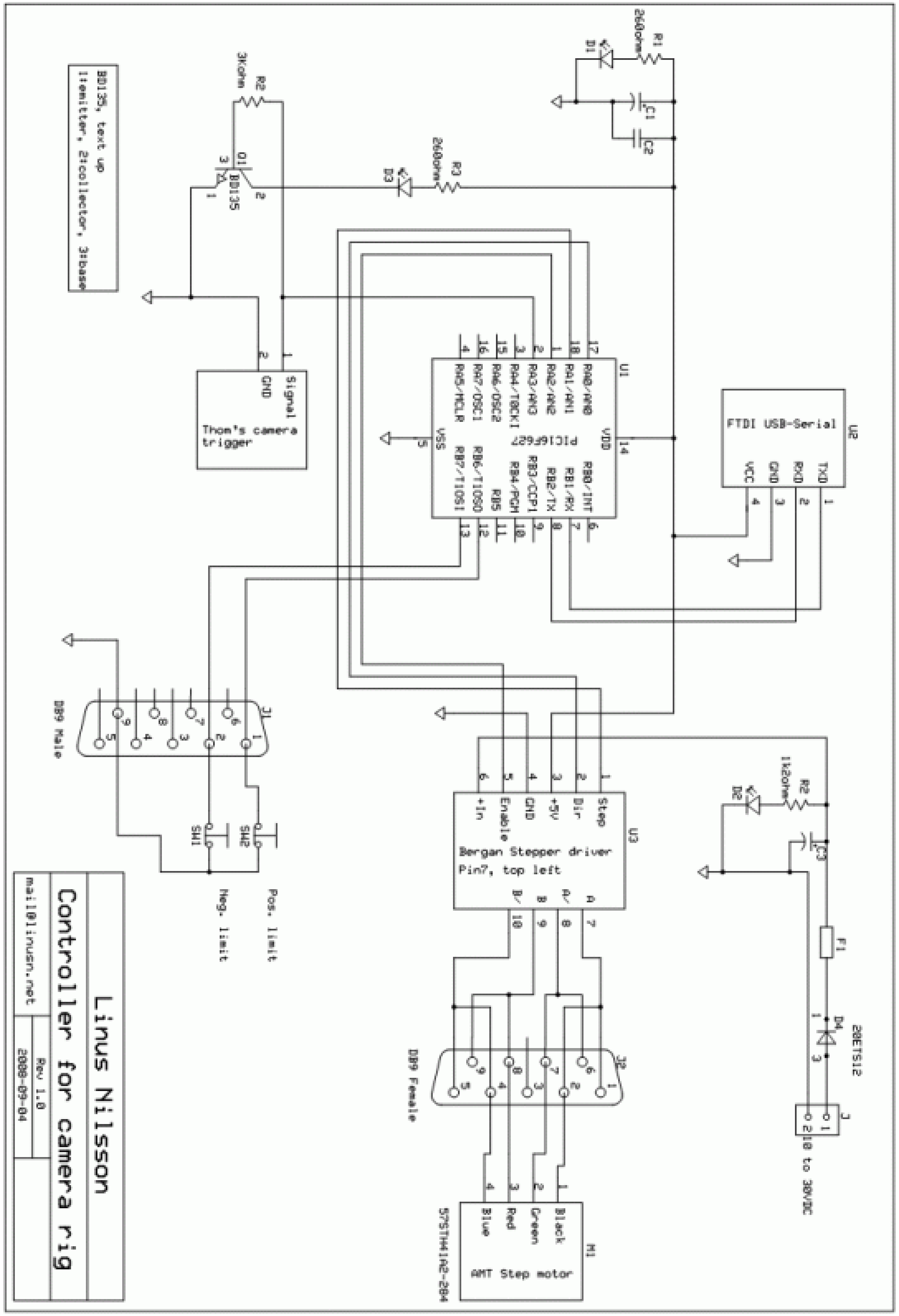
[Company Profile]

eXperience3D Sweden AB provides 3D viewing solutions to the Scandinavian market. We specialise in autostereoscopy, a method to see real 3D without glasses or extra accessories. As exclusive agents and technology partners of Spatial View Inc. and SeeFront Technologies GmbH, we represent the very best manufacturers of 3D products, and provide 3D content production and consulting services.

Appendix III - Part of the Property XML Schema

```
<!-- Root element -->
<xs:element name="property_names">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="property" type="propertyType"
                  minOccurs="1" maxOccurs="unbounded"></xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- All enteries for a property -->
<xs:complexType name="propertyType">
  <xs:sequence>
    <xs:element name="name" type="x_string"></xs:element>
    <xs:element name="edsdk_name" type="eds_string"></xs:element>
    <xs:element name="has_description" type="xs:boolean"></xs:element>
    <xs:element name="value" type="valueType"
                minOccurs="1" maxOccurs="unbounded"></xs:element>
  </xs:sequence>
</xs:complexType>

<!-- Stores the value -->
<xs:complexType name="valueType">
  <xs:sequence>
    <xs:element name="readout" type="x_string"></xs:element>
    <!-- Different data types-->
    <xs:choice>
      <xs:element name="hex" type="xs:hexBinary"></xs:element>
      <xs:element name="uint" type="xs:unsignedInt"></xs:element>
      <xs:element name="eds_enum" type="eds_string"></xs:element>
    </xs:choice>
    <xs:element name ="live_view" type="xs:boolean" minOccurs="0" maxOccurs="1">
    </xs:element>
  </xs:sequence>
</xs:complexType>

<!-- String with a minimum of 1 character -->
<xs:simpleType name="x_string">
  <xs:restriction base="xs:string">
    <xs:minLength value="1" />
  </xs:restriction>
</xs:simpleType>

<!-- String used to define a EDSDK name or enumeration -->
<xs:simpleType name="eds_string">
  <xs:restriction base="x_string">
    <xs:pattern value="kEds([a-zA-Z0-9_]+)" />
    <xs:maxLength value="100" />
  </xs:restriction>
</xs:simpleType>
```

Appendix IV - Circuit Diagram



Bibliography

Eckel, Bruce & Allison, Chuck (2004) *Thinking in C++*. Pearson Prentice Hall
Covington, Michael A. (2004) *Cable Release and Serial-Port Cable for Canon Digital Rebel Cameras*. Available: < http://www.covingtoninnovations.com/dslr/CanonRelease.html > (2008-05-14)
Crocker,Steve (2008) *Understanding C++ Exception Handling*. Available: < http://www.gamedev.net/reference/programming/features/cppexception/ > (2008-07-12)
(2008) *Why is it wrong to use std::auto_ptr<> with STL containers?*. Available: < http://stackoverflow.com/questions/111478/why-is-it-wrong-to-use-stdautoptr-with-stl-containers > (2008-08-16)
Williams, Anthony (2008) *Implementing a Thread-Safe Queue using Condition Variables (Updated)*. Available: < http://www.justsoftwaresolutions.co.uk/threading/implementing-a-thread-safe-queue-using-condition-variables.html > (2008-07-04)
D2XX Programmer's Guide(2006) Future Technology Devices International Ltd.
EDSDK 2.3 API Programming Reference(2008) Canon Inc.