# Eyetoy - EDA385 Project Report

Axel Ahlbeck - dat11aah@student.lu.se
Ricardo Gómez - soc14rgo@student.lu.se
Daniel Lundell - ada10dlu@student.lu.se
Erik Nimmermark - fpr05eni@student.lu.se

Instructor: Flavius Gruian

October 2015

**Abstract**

In the early 2000's, the EyeToy was released for the Playstation 2. It was a game where the player would stand in front of a camera and be visible on a monitor, and then different objects would show up on screen for the player to hit or avoid.

The idea behind this project was to mimic the EyeToy game using an FPGA-board and a camera. Because of the memory constraints, a lot of computations would have to be done in hardware and "on the fly" since image processing generally is computation-heavy. Both the hardware and the software needed to be constructed in such a way that memory usage was optimal, too keep within the memory- and time constraints.

The final product is far from perfect. Problems with the camera proved to be difficult to solve, resulting in an inferior picture quality and less functionality than expected.

# Contents

# 1 Introduction

The purpose of this project was to implement a clone of the popular EyeToy game released for the Playstation 2 in the beginning of this millennium. The game was unique because it used a camera with which to play, instead of a controller. Standing in front of the monitor, the player could see him- or herself on a monitor along with computer generated objects to either hit or avoid. There are many games for the EyeToy, but in this project, only the simplest possible game was to be developed - "Avoid the wall".

As the name suggests, the screen fills up with a "wall" with a cutout (a hole of some shape) for the player to fit into. If the player is able to squeeze into that cutout when the time is up, he or she succeeds and moves on to the next wall.

The final result differs quite a bit from the initial proposal, however in hardware more so than in software. To fetch the camera feed, a lot more (and more complex) hardware was needed. Initializing the camera had to be done in software, and with that an $I^2C$ had to be added. Also, the image processing required some extensive hardware (since it could not be done in software). Because of the above mentioned, the final result is more hardware-focused than originally planned.

# 2 Hardware Description

## 2.1 Architecture overview

An overview of the architecture can be seen in figure 1. The hardware consists of the Microblaze CPU, VGA controller, Image generation, Image processing and the PO2030N camera. The communication between the CPU and the PO2030N is done via an AXI bus. The VGA controller and Image generation generates the VGA signal to the monitor. The VGA controller is used for the timings and Image generation outputs the data bits through a VGA connector. Image processing is not integrated into the rest of the system it is therefore unconnected in figure 1.

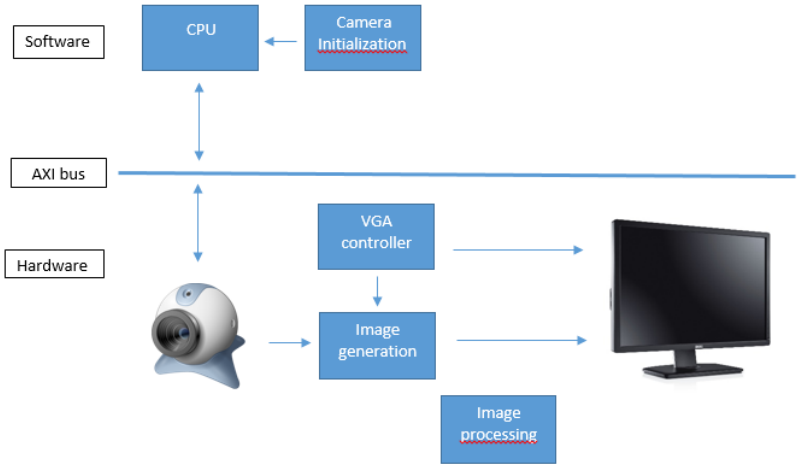


Figure 1: Architecture model

## 2.2 FPGA utilization

This is the utilization presented without any image processing present in the architecture:

- 12% of Slice Registers
- 32% of Slice LUTs

This is only the CPU with camera initialization, VGA controller and Image generation. This would of course be much higher if the image processing algorithms were in place.

## 2.3 PO2030N

The camera peripheral used in this project is PO2030N developed by PixelPlus. It has many features, including a 1/4.5 inch 640 x 480 active pixel array with color filter and micro-lens. It outputs 30 fps if clocked with 27MHz. The output format is 8-bit YCbCr by default (used in this project).

The clock feed into the camera is the 25Mhz pixel clock routed via an output DDR component(ODDR2)[1]. This is the only input to the camera except for the SCL and SDA which are used for the $I^2C$ communication. The hsync, vsync and pixel clock outputs coming from the camera are not used. The image data is output through eight data pins which are also connected to the system.

The output pixels are directly fed into the VGA (due to time constraints), which results in an image that has the correct shapes of everything but the colors are not pleasant to look at. The reason for this is that the VGA video

signal is specified to be RGB, so when YCbCr was output through the VGA connector it resulted in faulty coloration of the image. To combat this, the Cb and Cr values are simply ignored and the Y is sent to the R, G and B pins. Instead of the very unpleasant picture, the monitor displays a black and white picture where shapes and objects are much easier detected by the human eye.

## 2.4 VGA controller

The *vga_ctrl* block handles the feed and timing of the signals to the VGA connector. It has a pixel clock input, as well as the data output from the camera. For the timings a VGA controller [3] written by Ulrich Zoltán from digilent is used in conjunction with a custom-made image generation module to provide synchronization, as well as the RGB signals for the VGA. It also has a clock output that feeds a clock to the camera peripheral.

### 2.4.1 Img_Generation

*Img_generation* was originally only meant to check whether there is a wall or not at the current position and allow either the camera data feed or a wall pixel to the VGA connector. However, since the camera debugging took up so much time it also contains some timers and logic used for debugging and demonstration purposes.

## 2.5 Image Processing

In order to distinguish the player from other items or background different image processing algorithms were implemented. First of all, the user (the player in front of the camera) had to be extracted and separated from the background. For this reason, an algorithm targeting background extraction was needed. In the literature, different solutions are proposed. Among them, colour/brigthness background extraction was chosen.

### 2.5.1 Background Extraction

Constraining the background (i.e. a wall) to be either darker or brighter than the user enables simple background extraction by means of thresholding pixels depending on its brightness value. Considering the background to be brighter than the user (for example, if the wall is coloured white), comparing pixel by pixel the image to a brightness value, and generating a logical 0 or 1 on the output image depending on whether it is higher or lower than the threshold value will ideally lead to the required result. This result will be a binary image containing only 1s and 0s where the pixels containing the user will have value 1 and the pixels containing background will contain 0 (or vice versa).

A simple on-the-fly memory-less architecture can be directly deduced from the previous definition: pixels coming from the camera are processed in a combinational block that uses a comparator as decisor for background/foreground separation and generates 1s or 0s that form the output image.

### 2.5.2 Morphological post-processing

Whereas theoretically, the previous processing should be enough for proper background extraction, real-life scenarios show that post-processing of the binary output is required. The input images were constrained to images were the background was, for every pixel, brighter or darker than the foreground. However, this condition doesn't stand for real life images. Shot noise, or non-idealities of the background and foreground sections of the image (dark corners of the background, or bright clothes or parts of the body in the foreground) lead to noise in the resulting thresholded image. In this section, an image will be considered to be noisy if there are pixels that have binary values that differ from the desired output (an ideal background extraction). This undesired noise on the images can be solved by means of morphological processing operating over the output binary image.

Usually, noise in thresholded images can be assumed to be smaller (in pixel size) than the foreground images. This condition stands for most of the cases, where the noise sources are dark corners, or shot noise from the sensor. Where this condition stands, image processing algorithms called image erosion and dilation (and its combinations called image opening and closing) can be used to improve the quality of the output binary image. Detailed information about the definition and usage of this algorithms can be found in [4].

On-the-fly image processing was one of the main difficulties on the hardware implementation. As stated before, the lack of memory causes the necessity of an algorithm which is able to process the pixels at the same rate as they come from the camera. Hardware implementation was based on [5], written by Hugo Hedberg, Fredrik Kristensen and Viktor Öwall. However, small modifications were done to the paper's architecture, in order to be able to stop the whole processing chain in presence of an intermittent data flow.

As presented in [4], combination of different image dilation and erosion blocks leads to more complex processing blocks, which allows, for example, efficient binary noise reduction. For this reason, a top block containing any on-the-fly processing module and two FIFOs (one at the beginning and another one at the end) was designed. The purpose of this standardization was enabling quick modifications and concatenations of different processing stages. Figure 2 shows the high-level architecture of the processing block.

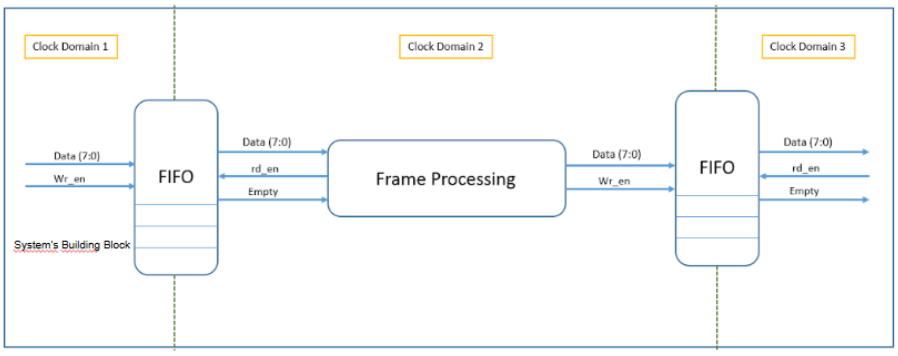


Figure 2: High-level architecture of the processing block.

#### 2.5.3 Hardware Integration and verification

After the design and verification of the previous blocks when being isolated, a real-life scenario was modelled and used for verification purposes. For this reason, a Matlab script was generated. This Matlab script takes an input image from the computer, and generates a text file with the pixel values row-wise, as it would be in a real-life scenario using the provided camera. This text file was used as stimuli for the hardware system, and as a result, another text file containing the output image was generated. This image was then verified by using a Matlab script that compares the simulated values with the expected ones in the theoretical model.

# 3 Software Description

The software is quite simple in its design. It handles the initialization of the camera settings. This is done by reading and writing to the registers in the camera via an AXI $I^2C$ bus. The camera settings are mostly unchanged from the default settings because most of them affect the camera timing signals, which were not used.

## 3.1 Camera Register Values

All register values except the ones specified in table 1 has the default value as stated in the "PO2030N *Preliminary* Data sheet"[2]. The only registers that were changed was the framesize values. The frameheight was set to 800 and framewidth was set to 525. This is the same values as the ones used in the counters for horizontal and vertical synchronization in the VGA controller. The "Value used" in Table 1 is the value of the register in question that was used in this project.

Table 1: Non-default camera registers

| Register address | Default value | Value used | Description |
|---|---|---|---|
| 04h | 03h | 03h | Framewidth HI |
| 05h | 83h | 20h | Framewidth LO |
| 06h | 01h | 02h | Frameheight HI |
| 07h | F3h | 0Dh | Frameheight LO |

### 3.2 Matlab Model

To effectively process the video stream, a Hardware system was implemented. However, this system requires some parameters, i.e. the threshold value. The impact of this parameters on the overall performance can be large. For this reason, a careful study of this parameters in order to obtain the desired performance is needed. However, studying this parameters directly on the final system would lead to large time consumption. For this reason, a Matlab model was implemented. This Matlab model uses commands and scripts from the Image Processing Toolbox, such as *imeerode* and *imdilate*, to model the expected behaviour of the system.

Input images were taken with the integrated camera of a laptop, and then uploaded to Matlab to be processed. The parameters to be set are: Threshold value, structuring elements' size (for morphological operations) and structuring elements' shape. This parameters were simulated and obtained for the specific size of the input image. However, when varying this size or the constraints of the video input (i.e. the brightness of the background or foreground), this values should be calculated again.

## 4 Installation and User Manual

### 4.1 Game description

Not playable, but the idea was to have the player stand in front of the camera and try to fit into the "holes" on the screen. If the player at the end of the level successfully managed to stand without touching the "wall", the player would pass that level. An example of a "wall" with the camera image in the "hole" can be seen in figure 3.

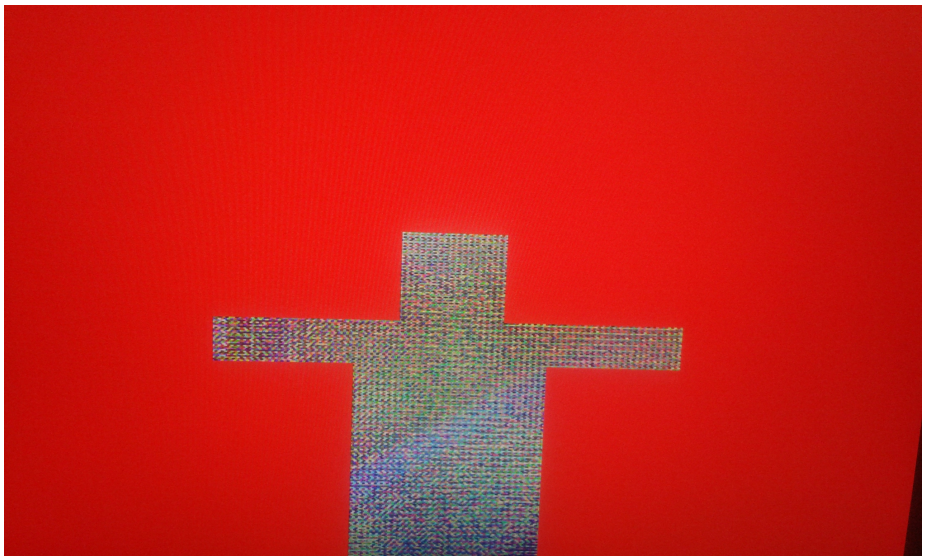

Figure 3: Example of a wall with camera feed in the "hole".

### 4.2 Connect camera and FPGA board

Connect the camera module to the PMOD connectors labled JA1 and JB1. Connect the Nexys3 FPGA board with two USB cables to a computer. Launch the Digilent Adept software and browse for the download.bit file in the uncompressed testcamnexys3\testapp\cpstreckrflaviusprojekt_hw_platform folder. Press program to load the bit file into the FPGA.

## 5 Problems Encountered

Many different problems were encountered over the course of this project. The biggest, however, was the problem with the camera.

### 5.1 Hardware

Apart from the camera issues the largest issue was the lack of memory on the Nexys-3 board. Since it did not have enough memory to buffer the video images, image erosion and dilation had to be implemented in hardware. This proved to be very difficult and time consuming to debug.

#### 5.1.1 Camera

Initially, the intention was to use a VmodCAM with the Nexys3-board. It was soon discovered that the VmodCAM was not compatible with the board, and

would require an Atlys-board instead. When there was no Atlys-board available, the solution was to use a different camera - PO2030N. The PO2030N was easily plugged into the Nexys3, but actually receiving the image was not straight-forward.

From initial tests, it seemed that there were no signals coming from the camera at all. Possibly, the signals were too weak to be detected or they were simply mapped wrong. To combat this, the design was reimplemented, the signals remapped and the VGA controller introduced. The mysteriously missing data signals as well as the SDA and SCL were now working correctly, but the hsync, vsync and pixel clock outputs coming from the camera still had no usable signals. These signals are currently not mapped, and the hsync and vsync from the VGA controller are used instead. Because of this there is a timing problem where the horizontal- and vertical synchronization signals as well as the pixel clock are not "in phase" with the data output from the camera, which causes errors in the video image as can be seen in figure 4.

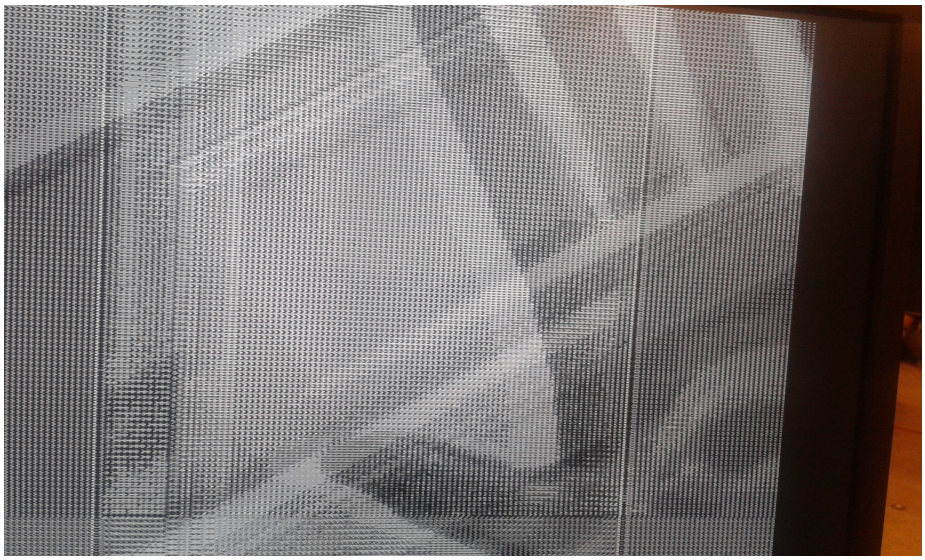

Figure 4: Distortions in the camera image.

## 5.2 Software

Software is only a small part of this project, but still not free from problems. Initializing the camera in software meant that values to the camera registers had to be sent over the AXI bus to the $I^2C$ and finally to the camera. The problem proved to be a combination of the signals from the camera not coming through properly and the libraries used in software not working in this particular design.

Instead of receiving a receiving the expected value from a register, the program would receive a 0, or be stuck in an endless loop waiting for the response. Eventually this was solved by using functions and addresses used in another project provided by Flavius.

# 6 Lessons Learned

When dealing with a project of this magnitude, it is important that everyone knows what changes has been made and what features has been added. For this, Git or any other revision control tool is great. Unfortunately, such a tool was not used for this project causing many changes to be lost.

Effectively dividing work is also important, and frequent meetups to discuss progress and setbacks. Do not underestimate the size and difficulty of a project. This project was almost a project of projects, in terms of how much had to be built in the end.

Finally, be very careful when mapping pins and signals in the xps. A single typo can cost a few days, just because of how hard errors can be to detect.

# 7 Contributions

- VGA-controller - Erik, Daniel
- Image generation - Erik
- Camera Initializing - Erik, Axel, Daniel
- Connecting Camera signals / routing - Erik, Axel, Daniel, Ricardo
- Matlab model - Ricardo
- Game logic - Erik, Axel
- Image processing - Ricardo
- $I^2C$ communication - Erik, Axel
- Report - Erik, Axel, Daniel
- Presentation - Axel, Daniel, Ricardo

# 8 References

[1] Spartan-6 FPGA SelectIO Resources. October 2015, `http://www.xilinx.com/support/documentation/user_guides/ug381.pdf`
Fetched October 2015

[2] *Preliminary* Data sheet PO2030N 1/4.5 Inch VGA Single Chip CMOS IMAGE SENSOR, 2005-10-14

[3] Ulrich Zoltán, VGA controller 2006-09-18. http://ece.wpi.edu/~rjduck/vga_controller_640_60.vhd

[4] Pierre Soille, "Morphological Image Analysis," Springer Berling Heidelberg, pp. 63-103, 2004.

[5] Hugo Hedberg, Fredrik Kristensen, and Viktor Öwall, "Low-complexity binary morphology architectures with flat rectangular structure elements," IEEE Trans. Circuits Syst.—Part I: Reg. Papers, vol. 55, no. 8, pp. 2216–2225, Aug. 2008.