Augmented Reality in a Virtual Reality Setup

The Camera

by

R. Van Gaalen
I. Zhang

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Thursday June 30, 2016 at 15:30.

Date:	June 15, 2016	
Students:	Robin van Gaalen	4148894
	Ian Zhang	4243943
Project duration:	April 18, 2016 – July 1, 2016	
Thesis committee:	Prof. dr. E. Eisemann,	TU Delft, supervisor
	Dr. ir. M. A. P. Pertijs,	TU Delft
	Dr. ing. I. E. Lager,	TU Delft

TU Delft

Abstract

Augmented Reality equipment is currently growing and showing a lot of promise. However, the equipment suffers from a few aspects: lag, range of vision and price. In this report an Augmented Reality system will be made by connecting two cameras through an FPGA to the VR system to cover all aspects that Augmented Reality suffers from.
While the project is currently an unfinished project, once the cameras work, the decrease in lag can be tested.

Contents

1 Introduction

Augmented Reality (AR) provides a way to blend the perceived reality with extra information, usually with computer generated input. While Head Mounted Displays (HMD) for AR use are developing quickly, there are quite a few restrictions that the current implementations suffer from.
The biggest restrictions of current Augmented Reality equipment are the lag and the range of vision. Most of the Augmented Reality HMDs solve the lag issue by using transparent material for the real-world environment, but suffer from range of the augmented vision. Google Glass, for example, only augments a small part of the view in the top-right corner of the right eye. [1] The Microsoft Hololens on the other hand does not suffer as much from the range. [2] Both do suffer from another important aspect: price. Google Glass sold for \$1500,- and Microsoft Hololens will be sold for \$3000,-.
A solution for the range problem and the price would be the use of Virtual Reality (VR) devices for Augmented Reality. Virtual Reality devices are defined by their ability to immerse the user into a virtual world. [3] The Oculus Rift DK1 was sold for \$300,- and the first Consumer Version is currently being sold for \$600,-. [4]
By connecting cameras to the VR device, it would be possible to get a fully immersive real-life experience combined with the extra information of the Augmented Reality, that would not suffer from range or price.
An issue with the combination of AR in a VR setup would be the delay. There have been earlier attempts of combining cameras with an Oculus in order to create AR, but since the videostream of the cameras is transmitted through the PC, the latency of the setup is quite high. [5] For this project, the Oculus Rift DK1 will be used, due to its direct availability.
A solution for the latency could be to, instead of transmitting the camera feed through the computer, transmit the camera feed through an ASIC (Application-Specific Integrated Circuit) or an FPGA (Field-Programmable Gate Array). Since these implementations are integrated circuits, a lot of data can be processed and transferred in a single tick of the clock. While ASICs are approximately 4 times faster than FPGAs, for this project, an FPGA will be used for the initial design, due to its flexibility with reprogramming. [6] Once the code has been thoroughly tested and is ready for mass-production, ASICs will be used.
In this project, two cameras will be attached to an Oculus Rift; these cameras will communicate with the FPGA and from the FPGA the camera feed will be sent to both the Oculus display and the PC. The PC will create an augmented view, that will then be sent through the FPGA to the Oculus.
As a proof of concept, the real-time feed will be augmented with IR vision. This will be done by attaching two additional cameras to the setup, and overlaying the IR vision on places that are too dark to see with the normal cameras. The entire setup can be found in Figure 1.1. [7] This report will only cover the Camera part of this project. The other parts of the project are split into two other theses that cover the PC part and the FPGA part.
In Chapter 2, the requirements of this project will be analyzed. Chapter 3 will explain which materials have been used and why they have been used. Chapter 4 will demonstrate the

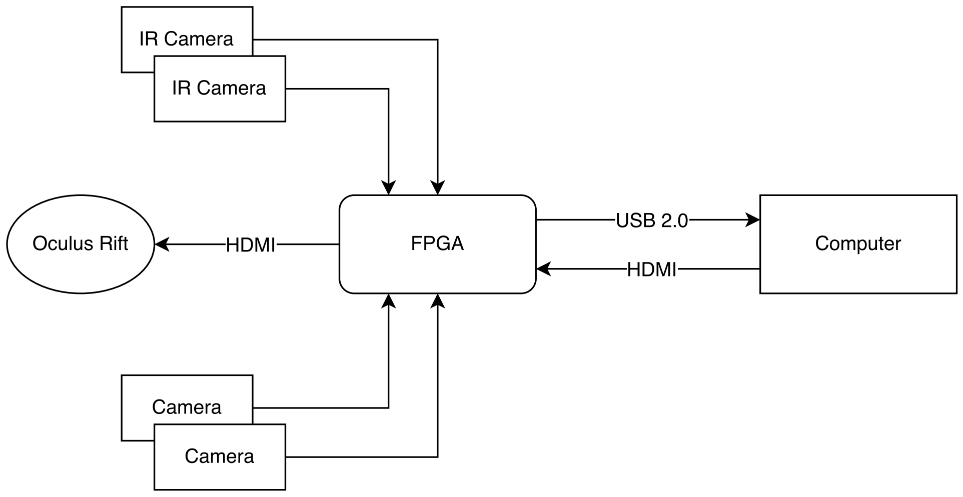


Figure 1.1: An overview of how the system communicates

different kinds of distortion that will have to be dealt with. Chapter 5 will descripe the I^{2}C protocol that is required for setting the camera up. In Chapter 6 a design of the final setup has been made using SolidWorks. Chapter 7 will show the results and Chapter 8 will discuss the difficulties that have been found on the way to the results. Finally in Chapter 9 the conclusions and a summary for future research can be found.

2 Programme of Requirements

2.1. Functional Requirements

- The product needs to be able to put a real-time video stream through to the Oculus Rift.
- The product needs to be able to augment the real-time video stream.
- The product needs to have a low enough latency and high enough frame rate, so that the user won't get simulator sickness
- The product needs to be a cheap alternative to other AR devices

Simulator sickness is a form of motion sickness that can occur when using the Oculus Rift. This has a high number of factors, but the most important factors for this project are latency and frame rate. When frames are dropped or when there is an input delay, the user can be discomforted. [8].
While the Oculus Rift Developer Center recommends a frame rate higher than the refresh rate (60 Hz), [9] a frame rate of 30 Hz should be enough for inexperienced users for virtual environments. [10]
The maximum price of the product depends on the competition. Google Glass might not be comparable, due to the range of the augmented view. Microsoft Hololens has a similar range, but costs $3000,-. The Meta 2 Development kit also has a similar range and only costs $949,-. [11] Therefore it must at least be lower than $949,-.

2.2. Environmental Requirements

Since this project currently mainly uses consumer hardware and connects them to each other, it should already fulfill the requirements.

2.3. System Requirements

- The FPGA needs a high clock frequency
- The FPGA needs enough GPIO pins to drive 4 cameras
- The FPGA needs an HDMI-in and HDMI-out port
- The cameras need to be able to output a high resolution and high frame rate
- The cameras need to be able to connect with the FPGA through GPIO
- Two of the four cameras must be able to see IR
- The computer must be able to run Unity3D
- The computer must have an HDMI-out port and USB-in

- Code must be written for FPGA to add 2 camera feeds together to send to the Oculus Rift
- Code must be written for the computer to read out incoming data and send out the augmented overlay
- The code must be optimized to have a low latency

As well as the functional requirements in Chapter 2.1, the system also needs to fulfill the seperate requirements for each part.
To minimize the latency, the FPGA must have a high frequency to process information faster and the code must be optimized for a higher throughput and therefore lower latency.
For connection of all seperate parts, the FPGA requires at least 64 GPIO pins to connect the four cameras with 16 GPIO pins each. It needs an HDMI-in to get an augmented view from the computer and an HDMI-out port to send the final view to the Oculus Rift.
The computer preferably needs to be able to run Unity3D, since this program will be used for most communication, and needs an HDMI-out port for communicating with the FPGA and USB to get data from the FPGA. USB is also required to upload code to the FPGA.

2.4. Manufacturing methodologies

In the case that the product is ready for mass production, the FPGA will be replaced with an ASIC. This way, the program will have a higher frequency, and it will also be cheaper to produce.
The item would probably be sold as an add-on for the VR system that is already bought, so the only things that need to be produced are the chip with the cameras.

2.5. Business strategies, marketing and sales opportunities

2.5.1. AR Add-on for VR systems

Since the current final product is probably to be sold as an add-on to a VR system, it can be sold to owners of VR with an HDMI input, who would like to make an AR system out of it.
The production cost of the product would be approximately €100,- per piece and it should probably be sold for approximately €200,- to remain as a cheap alternative to AR.
The market for this kind of product would be quite small. It targets owners of VR systems, which is already a small market and it is quite expensive. This kind of system is more suited for developers.

2.5.2. Full AR system

On the other side of the spectrum, it’s also highly possible to make an entire product out of it. In that case it would be embedded into a VR system. Making an own VR system would probably be recommended in that case, and achieving a production cost of approximately €300,- in total should be realizable.
Then it would be able to sell the product for approximately €700,- per piece, since it is still a cheaper alternative to AR systems.
The market would also be bigger than in the AR Add-on, since it targets every person that would buy either VR or AR systems.

3
Optical Materials

3.1. Cameras

Four cameras are to be used in this project. These cameras had to be selected based on the following requirements:

- Able to operate at 60 fps with a 720p resolution
- Cheap
- Able to output data, at least to some degree in parallel to decrease the bit-rate per data-line
- IR-filter easily removable
- Able to output uncompressed data (no JPEG) e.g. : YUV or RGB data
- Automatic corrections, such as gamma correction and automatic white-balance can be disabled.

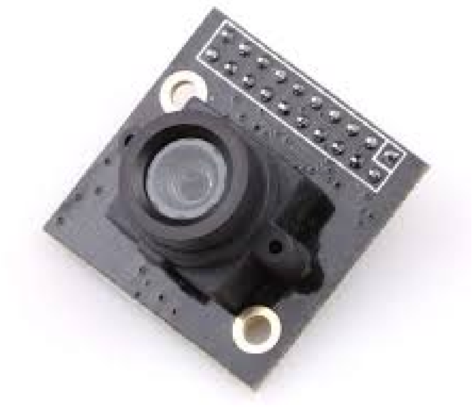
After some searching a camera was found that, at least in theory, satisfies all these requirements: the OV5642 camera module. An image of this camera module can be found in Figure 3.1a. An overview can be found in the datasheet. [12]
This camera can output data 8-bits parallel, the automatic corrections can all be disabled by setting certain registers and it's quite cheap (approximately €25,- per camera).
Because this camera has a lens which can be screwed off, and because the IR-filter is attached to the back part of the lens, this filter can easily be removed. Furthermore, since this lens can be removed, it is easy to fit it into the assembly in one of the camera holders, as shown in Chapter 6.1.2. The PCB has 2 screw holes that are to be used to fasten the camera to a camera holder.

3.2. Fish-eye lenses

The fish-eye lenses that will be put on top of the standard camera lenses are very simple novelty fish-eye lenses, meant to be put over mobile phone cameras. These lenses, while of seemingly low quality, correct for the Oculus' inherent distortion quite nicely. They were acquired at the local HEMA for 5 euros along with a phone clip, a macro lens and a wide lens. An image of the HEMA lenses can be found in Figure 3.1b

3.3. Beamsplitter

In order to make a setup with 4 cameras work, while not changing perspective when switching to the IR view, a beamsplitter is required. [13]
The beamsplitters will be made, just like the fish-eye lenses, with cheap novelty materials.


(a) The OV5642 camera module.

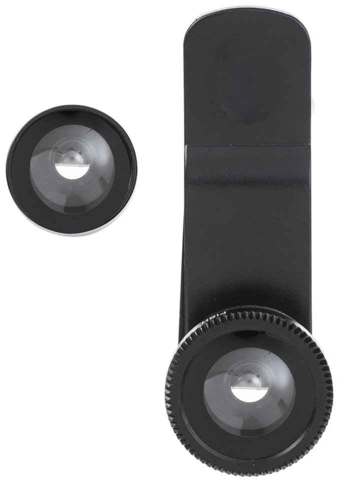

(b) The HEMA Fish-eye lens, can be pinned on a phone to take fish-eye photos.

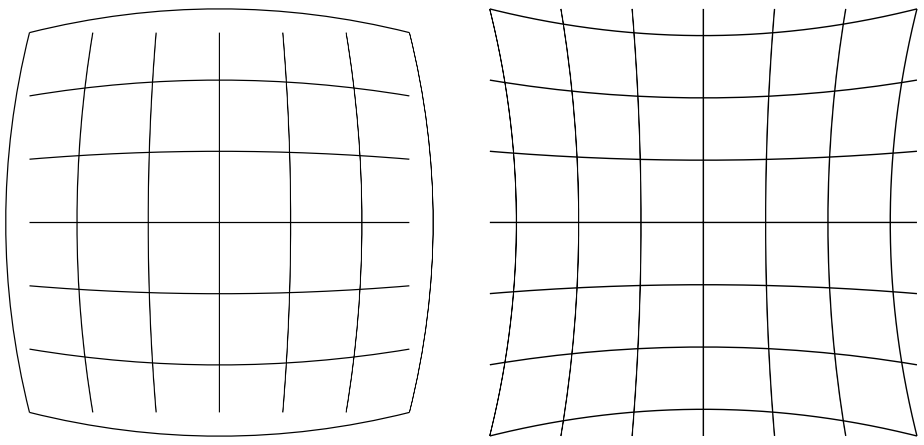
Figure 3.1: The used materials.

To make a beamsplitter, a half-mirror screenprotector, will be applied to a piece of PETG, a plastic polymer.
According to a DIY project found on the internet, this should render an almost 50-50 beamsplitter. [14]
The Screen cover that will be used is the so called ”pure mirror screenprotector” supposedly meant for women who want to touch up their make-up, though it makes for a surprisingly good beamsplitter.
The reason that both the lenses and the beamsplitter are improvised is that custom made lenses and beamsplitters are quite expensive. One quickly pays 500 euros for a single lens or beamsplitter.
It was elected that these make-do solutions are enough for our implementation.

4 Distortion

4.1. Barrel Distortion

Any unaltered image shown on the 7 inch Oculus display will become distorted with a *Pincushion Distortion* when viewed through the Oculus' lenses. [15] This is because the lenses were chosen for significant magnification, to create a very wide field-of-view.
The downside is that the image needs to be altered so it is not perceived as pincushion distorted. This is accomplished by applying an opposite distortion to the image: the *Barrel Distortion.* An image of both the the Barrel Distortion and the Pincushion Distortion can be found in Figures 4.1a and 4.1b. [16] [17]



(a) Barrel Distortion (b) Pincushion Distortion

Figure 4.1: Images of the types of distortion.

4.1.1. Applying the Barrel Distortion

There are several ways to create this barrel distortion: Through the FPGA, through the GPU or through lenses.

FPGA

One could implement a system on an FPGA that could buffer frames and apply a barrel distortion algorithm to them.
These systems have been created in the past, but from what could be gathered from these earlier experiments, it would be extremely difficult to make a system that could handle 4 video streams with a resolution of 720p and 60fps. [18]
It was chosen not to go for this approach for barrel distortion because it would be very difficult,

require a very expensive FPGA and would require some sort of buffer, thereby causing a delay which is to be avoided as much as possible.

GPU
The video stream from the camera could be sent to a computer which could apply a barrel distortion to the image before sending it to the FPGA. This would take away any purpose the FPGA has and create an unnecessary delay.
On the other hand, the overlays created in the computer can be distorted in this way, such that they would not have to be distorted later on in the FPGA.
It was chosen to barrel distort the overlay from the computer because it can be done easily there, and it would not cause a delay in the video streams from the cameras.

Lenses
Instead of creating convoluted electronic systems that attempt to properly correct the image at an as real time as possible frame rate, one could simply put a lens in front of the cameras that causes the desired barrel distortion. If one were to create perfectly matched lenses, the image could be corrected accurately.
These hypothetical ”perfect” lenses would be expensive and would need to be custom made. Instead of this perfect approach a financially more responsible approach was explored.
One type of lens that causes significant barrel distortion is a so called ”Fish-eye lens”. Generic Fish-eye lenses can be bought for as little as €5,-, and have the added advantage that they increase the field-of-view (FOV) of the cameras, which is a good thing, because the FOV of the Cameras was significantly lower than that of the human eye.(Humans possess a FOV of about 100 degrees to the left and 100 to the right, these fisheye lenses supposedly have a 110 degree FOV) [19]
By applying one of these novelty fish-eye lenses to a simple USB-web-cam, and viewing the video stream from said web-cam on the Oculus Rift, it was determined that, at least for this project, the distortion correction that these lenses yield is more than adequate: it is difficult to still notice the distortion.

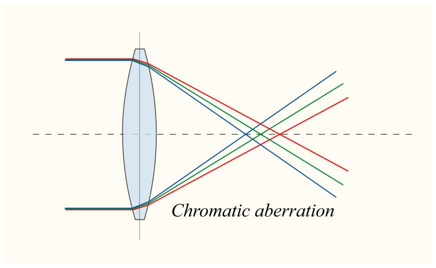
Conclusion
All in all: because the lenses are such a cheap and effective solution and yield the added advantage of increasing the FOV of the cameras, it was chosen to acquire 4 simple Fish-eye lenses which are to be applied to the camera to induce the proper barrel distortion.
However, if a future research group would like to continue the work delivered here, they could still opt for custom made lenses.

4.1.2. Calculating the distortion

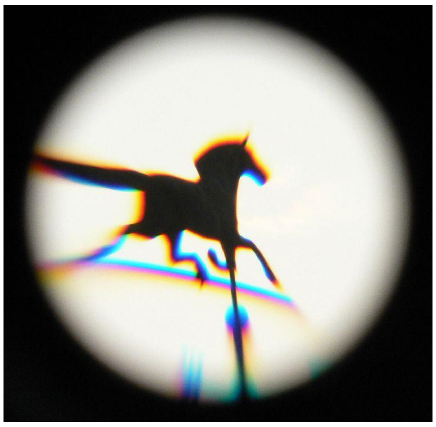
As stated in Chapter 4.1, to correct for the pincushion distortion created by the Oculus lenses, images shown on the Oculus' display need to be corrected for by applying a barrel distortion to them.
This is essentially an image transformation, which is characterized in Equation 4.1

$$Rnew = 1 + 0.22Rold^2 + 0.24Rold^4 \tag{4.1}$$

It is essentially a radial, or point-symmetric, transformation, in which each pixel is moved a certain distance away from the center of the image. That is to say: for each pixel one creates a vector, starting at the middle of the picture, ending at the pixels' location.
One takes the length of this vector as R_{old}, acquires R_{new} from the aforementioned formula, and multiplies the original vector by this R_{new}. This new vector will point to the new location for the pixel.
After this is has been done for all pixels, the image needs to be re-rendered, and a correctly transformed image will appear.
This formula has 3 non-zero coefficients: $k_1 = 1, k_2 = 0.22, k_3 = 0.24$. These coefficients are specific to the case of the Oculus lenses and setup, other lenses and other setups could yield different distortions, which would require different coefficients. But in the case of the Oculus, this is the transformation that would correct for the Oculus' lenses perfectly, and as such it is implemented on the computer to correct any rendered overlay sent to the FPGA.



(a) Illustration of different focal lengths and different wavelengths of light.



(b) Example picture of a chromatic aberration.

Figure 4.2: Illustrations of Chromatic Aberrations

4.2. Aberrations

Finally, the Oculus' lenses cause one more type of distortion, or rather: aberration, which leads to the user perceiving an ”incorrect” image. [15] This is what's called a ”chromatic aberration” and it's caused by the phenomena in which light of different wavelengths propagates at different speeds inside materials, which causes light at different wavelengths to be diffracted slightly differently; a phenomenon which can most easily be seen in a rainbow, or prism. [20]
Because light of different wavelengths, that is to say of different colours, is diffracted differently, the lenses in the Oculus have a slightly different ”focal length” for different colors of light.
It has a slightly larger focal length for light at the red end of the visible spectrum, thereby bending said light more towards the edges of the picture. It also has a slightly shorter focal length for light at the blue end of the spectrum, thereby pulling blue light more towards the middle of the picture. This is shown in Figure 4.2a. [21]
Consequently, the image becomes blurry and the colours seperate towards the edges of the picture, as shown in Figure 4.2b. [22]

It was chosen not to actively work towards correcting this phenomenon when it comes to the camera video streams, because this would have put too much strain on an already slightly ambitious project.
It is worth noting that the chromatic aberration was observed to be severely reduced by applying a fish-eye lens to a simple webcam. The chromatic aberration correction for the overlay stream can be handled by Unity3D/Oculus SDK.

5 I^2C

5.1. Introduction to I^2C

In order to configure the settings that the camera uses, it needs to be set using the I^2C (Inter-Integrated Circuit) protocol. I^2C is a multi-master protocol that can simultaneously connect any number of slaves, as long as they have different slave addresses.
The way the protocol works is best explained using a picture as seen in Figure 5.1 [23].
The I^2C protocol basically sends data over two lines: SCL (Serial Clock) and SDA (Serial Data). Since the data is sent serially, the protocol usually sends 8 bits and then an acknowledgment bit. The FSM will stay in the Ready state, until the ena signal is turned on. Once the ena signal is turned on, it will go to the Start state, and will start transmitting the given slave address bit by bit over the SDA port in the Command state.
Once it's done transmitting, it will ask for an acknowledgment bit from the slave. If it is transmitted, it will continue, if not it will also continue, but set an ack error flag so the user knows about this.
Then it will choose to either write or read, depending on the wr bit. To write, it will send the 8 data bits one by one over the SDA port. Then it will ask for another slave acknowledgment bit and will either restart in the Start state, if the ena bit is still on, or stop and return to the Ready state.
To read, it will read out 8 data bits one by one over the SDA port and will wait for a master acknowledgment bit and will either restart in the Start state, if the ena bit is still on, or stop and return to the Ready state.

5.2. Implementation of I^2C

For the implementation of the protocol, there are different kinds of platforms to set the registers. A somewhat complete list of the registers to be written can be found in the datasheet. [24]
There was also an implementation of the I^2C master in VHDL from the same source as the FSM image. The test code can be found in Appendix A. [23]

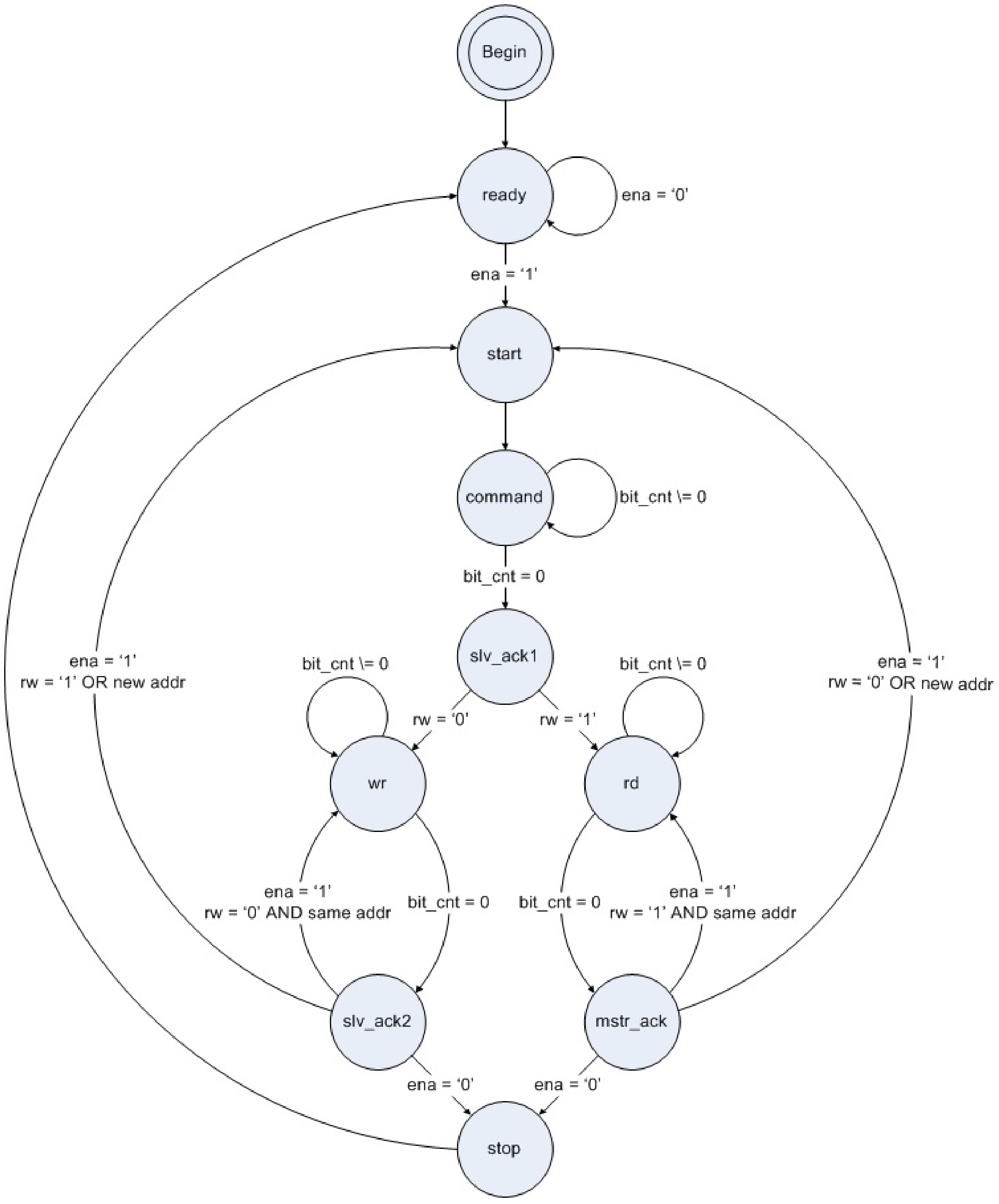


Figure 5.1: A Finite State Machine explaining the states of the I^2C protocol.

6 Assembly

The goal of this project is, at its essence, to create a Hybrid IR Night-Vision system.
The basic idea is to combine image feeds from 2 cameras: a normal camera and one from which the IR filter has been removed. Dark parts of the normal video stream may be supplemented by the IR video stream, thereby making all dark areas visible, while maintaining full photopic vision in the light areas.

6.1. SolidWorks Models

A system that contains the cameras and beamsplitters and is attachable to the Oculus needs to be created. This system needs to be lightweight, because of comfort, and it needs to be sturdy to keep everything aligned properly.
This system has been designed in Solidworks and visible in Figure 6.1 and is soon to be 3D-printed.

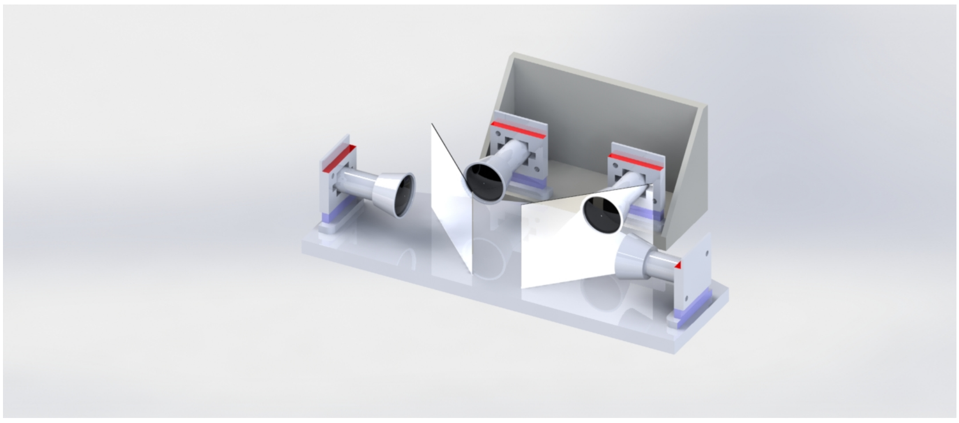


Figure 6.1: A SolidWorks model of the total setup

6.1.1. Camera

The model of the cameras can be found in Figure 6.2. This is a crude drawing of the cameras, with a fish-eye lens attached to it. These will be attached to camera holders so that they can be affixed to the base.

6.1.2. Camera holder

The model for the Camera holder can be found in Figure 6.3. This component will be attached to each of the cameras. Its purpose is to hold the cameras in place and attached to the base.

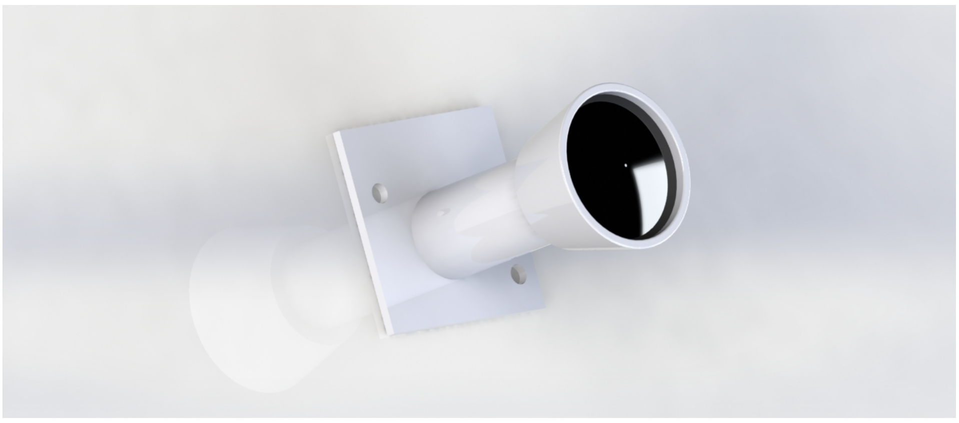


Figure 6.2: A SolidWorks model of the used camera.

It was made as a separate component, Instead of merging it with the base, so that it can be printed laying down (as shown in the picture). Otherwise the 3D printer would have had to fill each hole in the camera holder with filler material, which would then have to be removed, and, in practice, leads to less well defined and less accurate holes.
This component is to be inserted into various slots and glued.

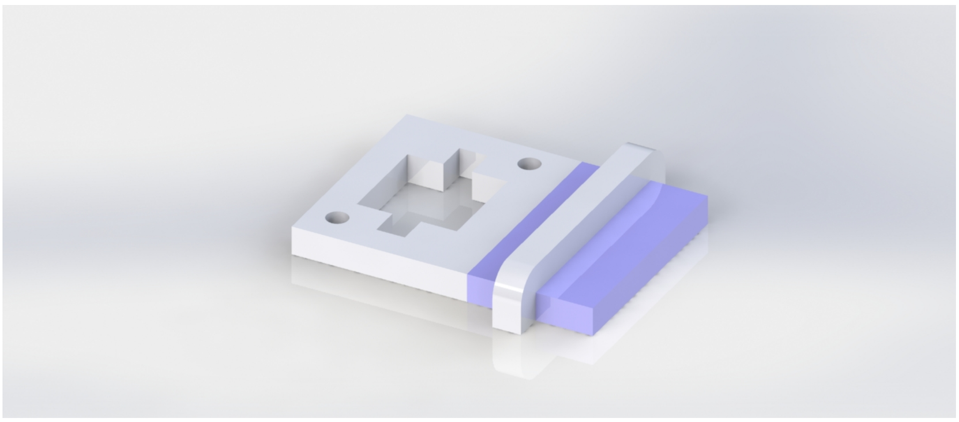


Figure 6.3: A SolidWorks model of the camera holder.

6.1.3. The Base

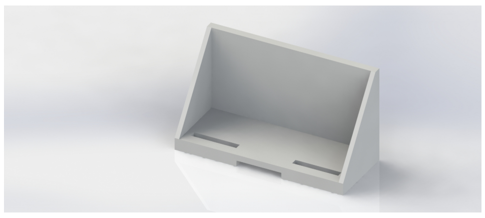
The base will hold the two main cameras and can be found in Figure 6.4a, The base will be attached to the Oculus with ”dual lock” Velcro tape, to make it easy to remove. Two camera holders are to be fitted in the two slots seen in Figure 6.4b.
A modular design approach was used, such that it is possible to only attach the base, without the IR add-on holding the IR cameras and beamsplitters from Chapter 6.1.4. The slot at the bottom can lock onto the IR add-on.

6.1.4. The IR add-on

The IR add-onholds the two IR cameras and can be found in Figure 6.5.
It will house two camera holders and two beamsplitters and can easily be attached to the base with two bolts. (The holes are not shown in render). The two slots on the sides are where two camera holders will go. The slots into which the beamsplitters are to be fitted are yet to be created but their positions will be shown later on.

6.1.5. Exploded View

The exploded view of the total model can be found in Figure 6.6.


(a) First part of the base without cameras

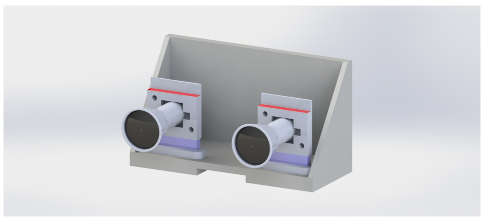

(b) A 2 camera implementation.

Figure 6.4: SolidWorks models of the first part of the base.

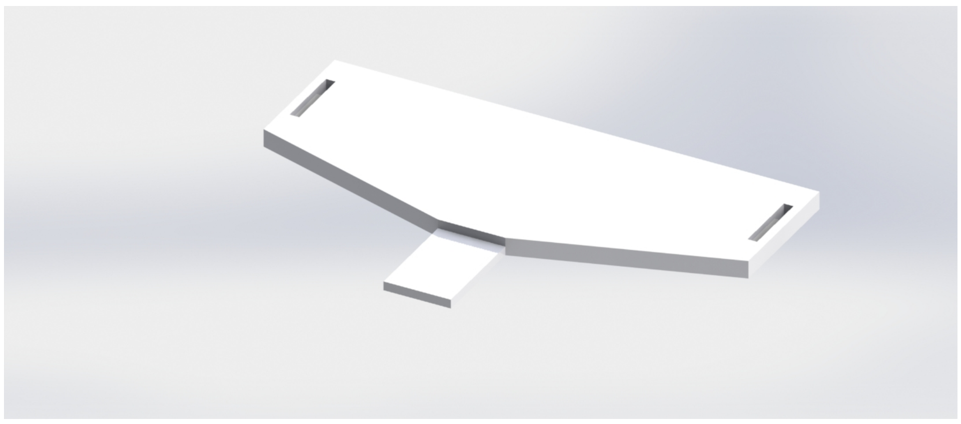


Figure 6.5: A SolidWorks model of the second part of the base.

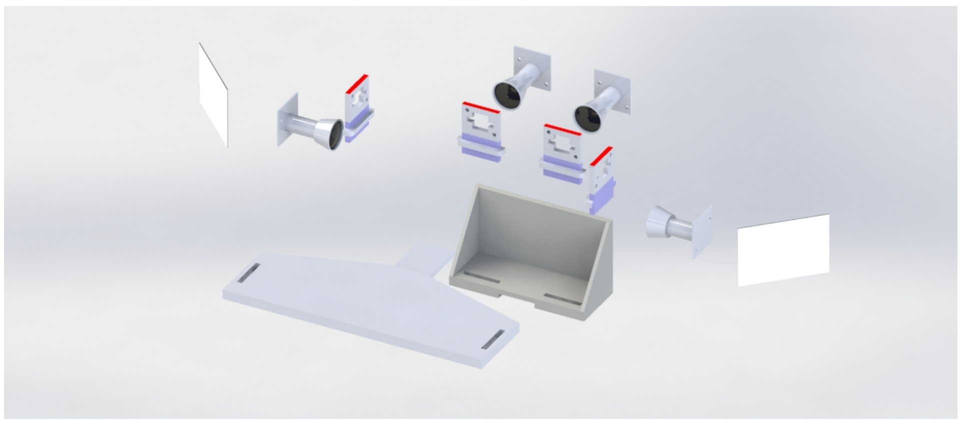


Figure 6.6: The exploded view of the total SolidWorks model.

7
Results

In this chapter, the results of the project will be discussed. The project is still a work in progress and therefore the results are incomplete.

7.1. Distortion

Barrel Distortion has successfully been applied using the fish-eye lens. The implementation of a Barrel Distortion has also successfully been done on the computer in Unity3D and this result can be found in Figure 7.1.

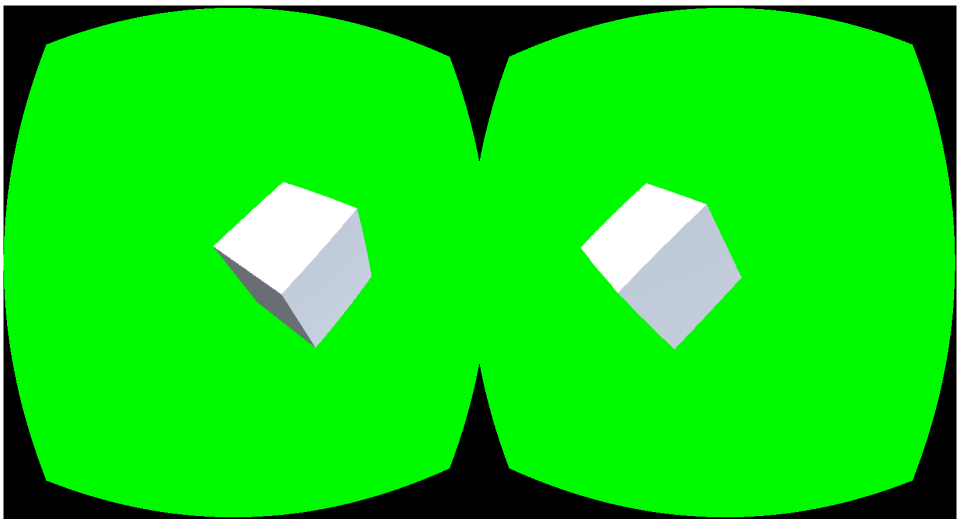


Figure 7.1: A Unity3D program, showing the utilization of Barrel Distortion and the preferred output to the Oculus.

7.2. I^2C

The results of writing the I^2C registers are more negative. While it is possible to write registers, currently a Raspberry Pi is being used to write them. Finding the correct I^2C registers to achieve the preferred 720p and 60 fps with an RGB output has proven to be a big challenge that is still to be overcome.
Currently tests are being held on displaying lower resolution and lower frame rate images with a YUV output. This results in synchronization problems, that are next on the list to be fixed.

7.3. Assembly

The total setup has yet to be made, but there should be no problems regarding the 3D-print of the setup.

8 Discussion

This chapter will discuss the mistakes that have been made and the difficulties that have been found on the way.

8.1. I^2C

8.1.1. The registers

Most of the registers that were to be used, can be found in the datasheet. [24] Just to write 720p, 11 pages full of registers had to be written. It might even be too risky to save all those registers to the FPGA. As stated, most, but not all registers written to in the pre-made I2C sequences can be found in the documentation. What this means is that we can't find out just what is happening exactly in these sequences, which makes it hard to change or learn from them. The manufacturer of the OV5642 has been emailed and requested to provide a working I2C sequence for our specifications. As of yet they have not responded.

8.1.2. FPGA

The primary idea of the I^2C implementation was to implement a code into the FPGA to write to the registers. The problem that occurred was that the FPGA code could not be synthesized by Vivado because the i2c_master.vhd uses inout ports and Vivado does not support them. However, the inout ports for SCL and SDA are required for the acknowledgment signals. For this reason the FPGA implementation of I^2C could not be used.

8.1.3. Raspberry Pi

As an alternative solution for writing the registers, a Raspberry Pi was used. The Raspberry Pi has multiple ways of writing and reading registers: through a python script or through the terminal.

Python

There are templates of code for writing and reading registers, [25] but this approach failed in practice.

Terminal

The Raspberry Pi has a light distribution of Ubuntu on it and could use the terminal to send commands to the camera. This would be done with the following command:

```
sudo i2cset —y 1 0x38 0x30 0x10
```

With 0x38 being the slave address, 0x30 being the register and 0x10 being the data to be written. However, the OV5642 does not have a 1-byte register, but a 2-byte register.
After long research, we found that the correct way to set the registers was to use the following style of commands:

```
sudo i2cset −y 1 0x38 0x30 0x00 0x10 i
```

With the 0x38 still being the slave address, the 0x30 combined with 0x00 being the 2-byte register and the data being 0x10.

8.1.4. The Colorspace

RGB

Even after finding out how to write the registers, the output of the camera was found to be gibberish. The expected data was in the form of 0x00, since the camera was covered with the lid. The result was a lot of different relatively high values.
By setting the registers to 720p, the output format was apparently set to JPEG. This needed to be changed to RGB. This option was not available in the datasheet though.

RGGB

The alternative was a raw RGGB output in a Bayer Pattern, which ended up being too slow to decode and could not be used in the end.

YUV

For the YUV, a decoder needed to be written. This has been done and it should work.

8.2. Synchronization

Currently, there are problems in the synchronization of the frames coming from the FPGA and the frames in the monitor. A real answer to this problem has yet to be found.

9 Conclusion

9.1. Conclusions

As follows from the results in Chapter 7, the project is still unfinished in a lot of ways. It is still impossible to display an image with the correct parameters to a screen through the FPGA, but these problems should be solved in the near future.
The use of Fish-eye lenses to solve the Barrel Distortion was a great success and saved a lot of computational problems and considerations.

9.2. Summary of Contributions

The most important contribution for creating AR in a VR setup would be the use of fish-eye lenses to handle the distortion. While a computer can probably do the barrel distortion with a better accuracy, the use of fish-eye lenses can significantly lower the latency.

9.3. Future Research

- Use of Fish-eye lenses is a great alternative to barrel distortion calculations
- The OV5642's documentation misses a lot of detail on when registers are written
- Writing to 2 byte registers in I^2C will require the use of I^2C block style commands.

A
Implementation of I^2C in VHDL

A.1. i2c_master.vhd

```
--
--   FileName:         i2c_master.vhd
--   Dependencies:     none
--   Design Software:  Quartus II 64-bit Version 13.1 Build 162 SJ Full
   Version
--
--   HDL CODE IS PROVIDED "AS IS."  DIGI-KEY EXPRESSLY DISCLAIMS ANY
--   WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT
--   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
--   PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL DIGI-KEY
--   BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL
--   DAMAGES, LOST PROFITS OR LOST DATA, HARM TO YOUR EQUIPMENT, COST OF
--   PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
--   BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF),
--   ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER SIMILAR COSTS.
--
--   Version History
--   Version 1.0 11/01/2012 Scott Larson
--     Initial Public Release
--   Version 2.0 06/20/2014 Scott Larson
--     Added ability to interface with different slaves in the same
   transaction
--     Corrected ack_error bug where ack_error went 'Z' instead of '1' on
   error
--     Corrected timing of when ack_error signal clears
--   Version 2.1 10/21/2014 Scott Larson
--     Replaced gated clock with clock enable
--     Adjusted timing of SCL during start and stop conditions
--   Version 2.2 02/05/2015 Scott Larson
--     Corrected small SDA glitch introduced in version 2.1
--


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY i2c_master IS
  GENERIC(
    input_clk : INTEGER := 50_000_000; --input clock speed from user logic
        in Hz
    bus_clk   : INTEGER := 400_000); --speed the i2c bus (scl) will run at
        in Hz
  PORT(
    clk       : IN     STD_LOGIC;     --system clock
    reset_n   : IN     STD_LOGIC;     --active low reset
    ena       : IN     STD_LOGIC;     --latch in command
    addr      : IN     STD_LOGIC_VECTOR(6 DOWNTO 0); --address of target
       slave
    rw        : IN     STD_LOGIC;     --'0' is write, '1' is read
    data_wr   : IN     STD_LOGIC_VECTOR(7 DOWNTO 0); --data to write to
       slave
    next_data : OUT    STD_LOGIC;     -- ask for next data to write
    busy      : OUT    STD_LOGIC;     --indicates transaction in progress
    data_rd   : OUT    STD_LOGIC_VECTOR(7 DOWNTO 0); --data read from slave
    ack_error : BUFFER STD_LOGIC;     --flag if improper acknowledge from
       slave
    sda       : INOUT  STD_LOGIC;     --serial data output of i2c bus
    scl       : INOUT  STD_LOGIC);    --serial clock output of i2c bus
END i2c_master;

ARCHITECTURE logic OF i2c_master IS
  CONSTANT divider : INTEGER := (input_clk / bus_clk) / 4; --number of
     clocks in 1/4 cycle of scl
  TYPE machine IS (ready, start, command, slv_ack1, wr, rd, slv_ack2,
     mstr_ack, stop); --needed states
  SIGNAL state         : machine;      --state machine
  SIGNAL data_clk      : STD_LOGIC;    --data clock for sda
  SIGNAL data_clk_prev : STD_LOGIC;    --data clock during previous system
     clock
  SIGNAL scl_clk       : STD_LOGIC;    --constantly running internal scl
  SIGNAL scl_ena       : STD_LOGIC               := '0'; --enables internal
     scl to output
  SIGNAL sda_int       : STD_LOGIC               := '1'; --internal sda
  SIGNAL sda_ena_n     : STD_LOGIC;    --enables internal sda to output
  SIGNAL addr_rw       : STD_LOGIC_VECTOR(7 DOWNTO 0); --latched in
     address and read/write
  SIGNAL data_tx       : STD_LOGIC_VECTOR(7 DOWNTO 0); --latched in data
     to write to slave
  SIGNAL data_rx       : STD_LOGIC_VECTOR(7 DOWNTO 0); --data received
     from slave
  SIGNAL bit_cnt       : INTEGER RANGE 0 TO 7 := 7; --tracks bit number in
       transaction
  SIGNAL stretch       : STD_LOGIC               := '0'; --identifies if
     slave is stretching scl
BEGIN

  --generate the timing for the bus clock (scl_clk) and the data clock (
     data_clk)
  PROCESS(clk, reset_n)
    VARIABLE count : INTEGER RANGE 0 TO divider * 4; --timing for clock
```

```
      generation
BEGIN
  IF (reset_n = '0') THEN          --reset asserted
    stretch <= '0';
    count   := 0;
  ELSIF (clk'EVENT AND clk = '1') THEN
    data_clk_prev <= data_clk;  --store previous value of data clock
    IF (count = divider * 4 - 1) THEN --end of timing cycle
      count := 0;               --reset timer
    ELSIF (stretch = '0') THEN  --clock stretching from slave not
       detected
      count := count + 1;       --continue clock generation timing
    END IF;
    CASE count IS
      WHEN 0 TO divider - 1 => --first 1/4 cycle of clocking
        scl_clk  <= '0';
        data_clk <= '0';
      WHEN divider TO divider * 2 - 1 => --second 1/4 cycle of clocking
        scl_clk  <= '0';
        data_clk <= '1';
      WHEN divider * 2 TO divider * 3 - 1 => --third 1/4 cycle of
         clocking
        scl_clk <= '1';      --release scl
        IF (scl = '0') THEN --detect if slave is stretching clock
          stretch <= '1';
        ELSE
          stretch <= '0';
        END IF;
        data_clk <= '1';
      WHEN OTHERS =>            --last 1/4 cycle of clocking
        scl_clk  <= '1';
        data_clk <= '0';
    END CASE;
  END IF;
END PROCESS;

--state machine and writing to sda during scl low (data_clk rising edge)
PROCESS(clk, reset_n)
BEGIN
  IF (reset_n = '0') THEN          --reset asserted
    state     <= ready;        --return to initial state
    busy      <= '1';          --indicate not available
    scl_ena   <= '0';          --sets scl high impedance
    sda_int   <= '1';          --sets sda high impedance
    ack_error <= '0';          --clear acknowledge error flag
    bit_cnt   <= 7;            --restarts data bit counter
    data_rd   <= "00000000";   --clear data read port
    next_data <= '0';
  ELSIF (clk'EVENT AND clk = '1') THEN
    IF (data_clk = '1' AND data_clk_prev = '0') THEN --data clock rising
        edge
      CASE state IS
        WHEN ready =>        --idle state
          IF (ena = '1') THEN --transaction requested
            busy    <= '1'; --flag busy
            addr_rw <= addr & rw; --collect requested slave address and
            command
          data_tx <= data_wr; --collect requested data to write
          state   <= start; --go to start bit
        ELSE            --remain idle
          busy  <= '0'; --unflag busy
          state <= ready; --remain idle
        END IF;
      WHEN start =>       --start bit of transaction
        busy    <= '1'; --resume busy if continuous mode
        sda_int <= addr_rw(bit_cnt); --set first address bit to bus
        state   <= command; --go to command
      WHEN command =>     --address and command byte of transaction
        IF (bit_cnt = 0) THEN --command transmit finished
          sda_int   <= '1'; --release sda for slave acknowledge
          bit_cnt   <= 7; --reset bit counter for "byte" states
          state     <= slv_ack1; --go to slave acknowledge (command)
        ELSE            --next clock cycle of command state
          bit_cnt <= bit_cnt - 1; --keep track of transaction bits
          sda_int <= addr_rw(bit_cnt - 1); --write address/command bit
              to bus
          state   <= command; --continue with command
        END IF;
      WHEN slv_ack1 =>    --slave acknowledge bit (command)
        IF (addr_rw(0) = '0') THEN --write command
          sda_int <= data_tx(bit_cnt); --write first bit of data
          state   <= wr; --go to write byte
        ELSE            --read command
          sda_int <= '1'; --release sda from incoming data
          state   <= rd; --go to read byte
        END IF;
      WHEN wr =>          --write byte of transaction
        busy <= '1';    --resume busy if continuous mode
        IF (bit_cnt = 0) THEN --write byte transmit finished
          sda_int <= '1'; --release sda for slave acknowledge
          bit_cnt <= 7; --reset bit counter for "byte" states
          next_data <= '1'; --ask for next data
          state   <= slv_ack2; --go to slave acknowledge (write)
        ELSE            --next clock cycle of write state
          bit_cnt <= bit_cnt - 1; --keep track of transaction bits
          sda_int <= data_tx(bit_cnt - 1); --write next bit to bus
          state   <= wr; --continue writing
        END IF;
      WHEN rd =>          --read byte of transaction
        busy <= '1';    --resume busy if continuous mode
        IF (bit_cnt = 0) THEN --read byte receive finished
          IF (ena = '1' AND addr_rw = addr & rw) THEN --continuing
              with another read at same address
            sda_int <= '0'; --acknowledge the byte has been received
          ELSE        --stopping or continuing with a write
            sda_int <= '1'; --send a no-acknowledge (before stop or
                repeated start)
          END IF;
          bit_cnt <= 7; --reset bit counter for "byte" states
          data_rd <= data_rx; --output received data
          state   <= mstr_ack; --go to master acknowledge
        ELSE            --next clock cycle of read state
```

```
          bit_cnt <= bit_cnt - 1; --keep track of transaction bits
          state   <= rd; --continue reading
        END IF;
      WHEN slv_ack2 =>    --slave acknowledge bit (write)
        next_data <= '0'; --stop asking for next data
        IF (ena = '1') THEN --continue transaction
          busy    <= '0'; --continue is accepted
          addr_rw <= addr & rw; --collect requested slave address and
             command
          data_tx <= data_wr; --collect requested data to write
          IF (addr_rw = addr & rw) THEN --continue transaction with
             another write
            sda_int <= data_wr(bit_cnt); --write first bit of data
            state   <= wr; --go to write byte
          ELSE        --continue transaction with a read or new slave
            state <= start; --go to repeated start
          END IF;
        ELSE            --complete transaction
          state <= stop; --go to stop bit
        END IF;
      WHEN mstr_ack =>    --master acknowledge bit after a read
        IF (ena = '1') THEN --continue transaction
          busy    <= '0'; --continue is accepted and data received is
             available on bus
          addr_rw <= addr & rw; --collect requested slave address and
             command
          data_tx <= data_wr; --collect requested data to write
          IF (addr_rw = addr & rw) THEN --continue transaction with
             another read
            sda_int <= '1'; --release sda from incoming data
            state   <= rd; --go to read byte
          ELSE        --continue transaction with a write or new slave
            state <= start; --repeated start
          END IF;
        ELSE            --complete transaction
          state <= stop; --go to stop bit
        END IF;
      WHEN stop =>        --stop bit of transaction
        busy  <= '0';   --unflag busy
        state <= ready; --go to idle state
    END CASE;
  ELSIF (data_clk = '0' AND data_clk_prev = '1') THEN --data clock
      falling edge
    CASE state IS
      WHEN start =>
        IF (scl_ena = '0') THEN --starting new transaction
          scl_ena   <= '1'; --enable scl output
          ack_error <= '0'; --reset acknowledge error output
        END IF;
      WHEN slv_ack1 =>    --receiving slave acknowledge (command)
        IF (sda /= '0' OR ack_error = '1') THEN --no-acknowledge or
           previous no-acknowledge
          ack_error <= '1'; --set error output if no-acknowledge
        END IF;
      WHEN rd =>          --receiving slave data
        data_rx(bit_cnt) <= sda; --receive current slave data bit
```

```
          WHEN slv_ack2 =>    --receiving slave acknowledge (write)
            IF (sda /= '0' OR ack_error = '1') THEN --no-acknowledge or
                previous no-acknowledge
              ack_error <= '1'; --set error output if no-acknowledge
            END IF;
          WHEN stop =>
            scl_ena <= '0'; --disable scl
          WHEN OTHERS =>
            NULL;
        END CASE;
      END IF;
    END IF;
  END PROCESS;

  --set sda output
  WITH state SELECT sda_ena_n <=
    data_clk_prev WHEN start,       --generate start condition
    NOT data_clk_prev WHEN stop,    --generate stop condition
    sda_int WHEN OTHERS;            --set to internal sda signal

  --set scl and sda outputs
  scl <= '0' WHEN (scl_ena = '1' AND scl_clk = '0') ELSE 'Z';
  sda <= '0' WHEN sda_ena_n = '0' ELSE 'Z';

END logic;
```

A.2. i2c_writer.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 03.06.2016 10:46:13
-- Design Name:
-- Module Name: i2c_writer - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
```

```
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity i2c_writer is
  Port(busy      : in  STD_LOGIC;
      ack_error : in  STD_LOGIC;
      reset_n   : in  STD_LOGIC;
      next_data : in  STD_LOGIC;
      start     : in  STD_LOGIC;
      clk       : in  STD_LOGIC;
      addr_out  : out STD_LOGIC_VECTOR(6 downto 0);
      data_wr   : out STD_LOGIC_VECTOR(7 downto 0);
      ena_out   : out STD_LOGIC;
      led       : out STD_LOGIC;
      led2      : out STD_LOGIC;
      rw_out    : out STD_LOGIC);
end i2c_writer;

architecture Behavioral of i2c_writer is
  type state_type is (start_state, write_state, done_state,
     ack_error_state);
  type wrarray is array (0 to 3) of STD_LOGIC_VECTOR(7 downto 0);
  signal state       : state_type;
  signal writearray  : wrarray := (X"30", X"11", X"09", X"00");
  signal count       : integer := 0;
  signal ena         : STD_LOGIC;
  signal addr        : STD_LOGIC_VECTOR(6 downto 0);
  signal rw          : STD_LOGIC;
  signal ack_error_s : std_logic;
begin
  process(reset_n, clk, next_data)
  begin
    if (reset_n = '0') then
      state       <= start_state;
      count       <= 0;
      ena         <= '0';
      addr        <= "0000000";
      rw          <= '0';
      ack_error_s <= '0';
    else
      if rising_edge(next_data) then
        count <= count + 1;
        if state = start_state then
            count <= 0;
        end if;
      end if;
      if rising_edge(clk) then
        case state is
          when start_state =>
            if start <= '1' then
              state <= write_state;
              ena   <= '1';
              addr  <= "0111100";
            rw    <= '0';
          else
            state <= start_state;
            ena   <= '0';
            rw    <= '0';
          end if;
        when write_state =>
          addr <= "0111100";
          case count is
            when 0 =>
              ena <= '1';
              rw  <= '0';
            when 1 =>
              ena <= '1';
              rw  <= '0';
            when 2 =>
              ena <= '1';
              rw  <= '0';
            when others =>
              ena <= '0';
              rw  <= '0';
          end case;

          if count = 3 then
            if ack_error = '1' then
              state <= ack_error_state;
            else
              state <= done_state;
            end if;
          else
            state <= write_state;
          end if;
        when done_state =>
          state <= start_state;
        when ack_error_state =>
          ack_error_s <= '1';
          if start = '1' then
            state <= start_state;
          else
            state <= ack_error_state;
          end if;
      end case;
    end if;
  end if;
end process;
led2    <= NOT ena;
led     <= ack_error_s;
data_wr <= writearray(count);
ena_out <= ena;
addr_out <= addr;
rw_out <= rw;
end Behavioral;
```

Bibliography

[1] "Tech specs - Google Glass Help," [Online]. Available: https://support.google.com/glass/answer/3064128?hl=en, (Accessed on 09/06/2016).

[2] "Microsoft hololens hardware," https://www.microsoft.com/microsoft-hololens/en-us/hardware, (Accessed on 09/06/2016).

[3] Y. Boas, "Overview of virtual reality technologies," in *Interactive Multimedia Conference. Southampton*, 2013.

[4] "Oculus – shop," [Online]. Available: https://shop.oculus.com/en-us/cart/, (Accessed on 10/06/2016).

[5] M. Höll, N. Heran, and V. Lepetit, "Augmented Reality Oculus Rift," [Online]. Available: http://arxiv.org/pdf/1604.08848.pdf, (Accessed on 10/06/2016).

[6] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[7] R. Blokker and L. Noordam, "Augmented reality using a virtual reality setup, implementation on an FPGA device," Jun. 2016.

[8] "Developer center — documentation and sdks | oculus," [Online]. Available: https://developer.oculus.com/documentation/intro-vr/latest/concepts/bp_app_simulator_sickness/, (Accessed on 13/06/2016).

[9] "Developer center — documentation and sdks | oculus," [Online]. Available: https://developer.oculus.com/documentation/intro-vr/latest/concepts/bp_intro/, (Accessed on 13/06/2016).

[10] J. Y. C. Chen and J. E. Thropp, "Review of Low Frame Rate Effects on Human Performance," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 37, no. 6, pp. 1063–1076, Nov. 2007.

[11] "Augmented reality | meta company," [Online]. Available: https://www.metavision.com/buy, (Accessed on 13/06/2016).

[12] "Ov5642 datasheet," [Online]. Available: http://www.uctronics.com/download/cam_module/1Inch4_5_Megapixel_OV5642_CMOS_Camera_Module_DS_V1.1.pdf, (Accessed on 13/06/2016).

[13] Francis A. Jenkins & Harvey E. White, *Fundamentals of Optics*. [Online]. Available: http://archive.org/details/FundamentalsOfOptics

[14] "Building a very cheap DIY beam splitter 3D camera rig - 3D Vision blog," [Online]. Available: http://3dvision-blog.com/3983-building-a-very-cheap-diy-beam-splitter-3d-camera-rig/, (Accessed on 13/06/2016).

[15] "Developer center — documentation and sdks | oculus," [Online]. Available: https://developer.oculus.com/documentation/pcsdk/0.5/concepts/dg-render/, (Accessed on 13/06/2016).

[16] WolfWings, "Barrel distortion visual example," Sep. 2008. [Online]. Available: https://commons.wikimedia.org/wiki/File:Barrel_distortion.svg

[17] ——, “Pincushion distortion visual example,” Sep. 2008. [Online]. Available: https://commons.wikimedia.org/wiki/File:Pincushion_distortion.svg

[18] H. Blasinski, W. Hai, and F. Lohier, “FPGA architecture for real-time barrel distortion correction of colour images,” in *2011 IEEE International Conference on Multimedia and Expo*, Jul. 2011, pp. 1–6.

[19] P. J. Savino and H. V. Danesh-Meyer, *Color Atlas and Synopsis of Clinical Ophthalmology – Wills Eye Institute – Neuro-Ophthalmology*. Lippincott Williams & Wilkins, May 2012.

[20] D. H. Marimount and B. A. Wandell, “Matching color images: the effects of axial chromatic aberration,” *Journal of the Optical Society of America A: Optics and Image Science, and Vision*, vol. 11, no. 12, pp. 3113–3122, 1994.

[21] “File:chromatic aberration lens diagram.svg - wikimedia commons,” [Online]. Available: https://commons.wikimedia.org/wiki/File:Chromatic_aberration_lens_diagram.svg, (Accessed on 14/06/2016).

[22] “How to correct chromatic aberration with vegas? : MAGIX vegas,” [Online]. Available: https://forums.creativecow.net/thread/24/900900, (Accessed on 14/06/2016).

[23] “I2C Master (VHDL) - logic - eewiki,” [Online]. Available: https://eewiki.net/pages/viewpage.action?pageId=10125324, (Accessed on 13/06/2016).

[24] “OV5642 Camera Module Software Applications,” [Online]. Available: http://forum.motofan.ru/index.php?act=attach&type=post&id=256766, (Accessed on 13/06/2016).

[25] “Using the I2C Interface – Raspberry Pi Projects.” [Online]. Available: http://www.raspberry-projects.com/pi/programming-in-python/i2c-programming-in-python/using-the-i2c-interface-2