Novel use and Holistic Development of Inexpensive Field Programmable Gate Array (FPGA) for Low Latency Streaming Augmented Reality Systems

Douglas A. J. Brion, Thomas R. B. Carotti, David E. Shah

Word Count (excluding appendices): 7504
Final Report submitted in partial fulfilment of the requirements for the end of year EIE research project at Imperial College London.

Group 12:
Douglas Brion, Thomas Carotti, David Shah
May 2016
Contents

Abstract 4

1 Introduction and Meeting 7
   1.1 Meeting with Dr Bouganis ............................... 7
   1.2 Shifts from original plan ............................... 7
   1.3 Roadmap .............................................. 8

2 Project Management 9
   2.1 Progress since Interim Report .......................... 9
   2.2 Design Specification .................................. 9
      2.2.1 Hardware Specification ............................. 9
      2.2.2 Elastic C Specification ............................ 10
      2.2.3 AR Minecraft Specification ......................... 10
   2.3 Design Process ......................................... 11
   2.4 Proposed and Selected Solutions ........................ 12
   2.5 Planning and Time Management .......................... 12

3 Elastic C Design 14
   3.1 The Compiler .......................................... 14
   3.2 Features .............................................. 14
   3.3 Elastic C Syntax ...................................... 15
   3.4 Principles of Operation ............................... 16
   3.5 Lib Elastic ........................................... 17
   3.6 Atom Package ......................................... 18

4 PCB Design 19
   4.1 Version 1.0 ............................................ 19
   4.2 Version 1.1 ............................................ 20
5 Driver Development

6 Headset Model Design
   6.1 Ergonomics and Anthropometrics
   6.2 Optics
   6.3 Additive Manufacturing using Filament Deposition Modelling

7 Marker Recognition
   7.1 Marker Design
   7.2 Recognition Algorithm

8 Matrix Transformations
   8.1 The Aim
   8.2 Mathematical Streaming Implementation

9 Minecraft Gameplay
   9.1 Block Breaking
   9.2 Block Placing
   9.3 Inventory
   9.4 Maps
   9.5 Texture Packs

10 HDMI Input
   10.1 Implementation
   10.2 Endless Possibilities

11 Evaluation
   11.1 Performance Analysis
   11.2 Summary of Project
      11.2.1 Project Photos
      11.2.2 FPGA Resource Usage
      11.2.3 Timing Performance
      11.2.4 Project Cost
   11.3 Knowledge obtained
Dedication

We would like to thank Dr Christos Bouganis for his effervescent enthusiasm in our project, as well as offering his expertise and assistance whenever needed.
Abstract

In this paper we discuss the use of an inexpensive FPGA (in our case a Cyclone III EP3C16) as a tool for creating low latency streaming augmented reality systems. We will also review the development of our own simple tool for the rapid prototyping of streaming image processing systems. The creation of our PCB and Headset shall be explained as well as the methods we used for our marker recognition and streaming matrix perspective transformations for realistic image overlaying. Finally the foundation of basic game play features of AR Minecraft will be discussed followed by how we plan to continue this project.
Chapter 1

Introduction and Meeting

1.1 Meeting with Dr Bouganis

A meeting with Dr Bouganis was organised in order to discuss the current progress in the project and what was planned for the future. At this stage, due to working over easter, an AR demo on the headset, doing image processing effects on the world, had already been created. These included: raw, sobel, canny, sepia and emboss. The frame rates achieved were very pleasing with rates of $\sim 90$Hz stably and up to $\sim 150$Hz in experimentation. After Dr Bouganis saw this demonstration questions were raised about the planned circuit goggles concept which would allow users to look at a digital circuit while the headset solved the output. After a long discussion it was decided that this project was not ambitious enough nor did it utilise the full capability of the technologies that had already been developed. Dr Bouganis had a desire to play Doom in AR around the EEE department and to also use it for drones. His enthusiasm for the headset inspired the team to aim for loftier goals. Thus the end goal became to overlay Minecraft on the walls around the user.

1.2 Shifts from original plan

As mentioned the previous plans for the project involved creating a digital circuit simulator allowing users to simply ‘look’ at a circuit on a grid for it to be solved. This, after closer thought, seemed almost unrelated to the AR/VR headset which had been created so far as it didn’t utilise the ability to create a 3d world around you, nor was it immersive at all. The suggestion by Dr Bouganis to overlay Doom textures on the real world inspired the team. Thusly the project goal shifted and there was now a desire to create a game in AR.

Enter Minecraft in AR. The plan: overlay 2D Minecraft blocks on the walls around the user. Walls will be picked up by the cameras which recognise markers that have been placed on
the desired surface. Mining and placing of blocks will also be possible as well as live matrix transformations in order for the Minecraft blocks to match the perspective of the wall.

1.3 Roadmap

The Roadmap for this project is to get the following working in the order listed below depending on the time constraints:

1. Marker Detection reliably working
2. Texture Rendering
3. Matrix creation from marker position
4. Texture Transform
5. Block manipulation
6. Multiple Wall detection
7. HDMI input
Chapter 2

Project Management

2.1 Progress since Interim Report

A significant amount of progress has been made since the interim report, particularly as the group put in a lot of work over Easter. Therefore the following have already been achieved:

- Designed, ordered and received version 1.1 of the PCB
- Finished the drivers, in particular improving the image quality
- Developed a tool, Elastic C, that converts a C-like language to a single-cycle pipelined (or, more experimentally a state machine based multicycle) VHDL implementation.
- Significantly changed the specification, from Circuit Goggles to AR Minecraft
- Developed AR Minecraft, applying perspective transforms to the image based on the detected position of multiple markers

2.2 Design Specification

The project consists of 3 distinct parts. As such, a separate specification for each has been provided.

2.2.1 Hardware Specification

The AR headset being developed should:
1. Be comfortable to wear for extended periods of time, for as many people as possible (see section 6.1 - ‘Ergonomics and Anthropometrics’ for more detail)

2. Have a minimum resolution of 512x480 per eye, and support a frame rate of at least 80fps (ideally 90fps)

3. Feature the EP3C16 FPGA, as used on the DE0

4. Have a total parts cost of less than £300

5. Feature buttons to allow user interaction

### 2.2.2 Elastic C Specification

The Elastic C tool should, at a minimum:

1. Convert a C-like language to a pipelined single-cycle VHDL implementation accepted by Quartus

2. Feature variable width signed and unsigned types; if statements; and support static analysis of for loops

3. Support the generation line buffers to create moving windows (these will be called stream2ds) used for convolution and other image processing tasks

4. Insert pipeline registers automatically, using a simple timing model specifically for the EP3C16 and a user specified clock frequency

5. Take less than 30 seconds to compile simple image processing tasks such as Sobel or Canny edge detection

These features are discussed further in Chapter 3

### 2.2.3 AR Minecraft Specification

1. Recognise four multicolour markers, and using their positions to determine the transformation matrix needed to match the perspective of the markers

2. Render a 2D grid of blocks with different multicolour textures, having the same perspective as the markers

3. Allow the user to place and break blocks
4. Keep track of an inventory, incrementing when the user breaks a block and decrementing when the user places one

5. Support multiple different maps, which are triggered using different combinations of marker colours

### 2.3 Design Process

Instead of using Catapult-C, all of the streaming parts of the project will be implemented in a language developed for image processing, Elastic C. A Nios II/e softcore will also be used for the less time critical parts (which require floating point).

It is believed that this tool, Elastic C, has a faster design work flow and is significantly easier to use, with only a small reduction in final performance. One of the main features that was disliked about Catapult-C was the unnecessarily bulky IDE. Constantly switching windows to unroll loops was felt to be a waste of time, when fundamentally all that was needed were the output files. Users of Elastic-C are also presented with the freedom of using any editor they please, in the groups case, Atom, with a package for syntax highlighting and auto completion that the members developed. (see section 3.6).

A simplified block diagram of the final image processing system is shown below. The blocks and their purpose will be explained in greater detail throughout the report.

![Figure 2.1: Block Diagram of Image Processing System](image_url)
2.4 Proposed and Selected Solutions

First of all, one had to consider how to approach rendering the blocks in a realistic way. The first idea was to always render them as flat; with no scaling, perspective or transformation. However, it was decided that this would not provide a very immersive AR experience. Thus different ways were considered to apply a perspective transformation to the image in order to map it to a surface. The next idea was to apply the transformation in a non-real-time manner, outputting the transformed image to SDRAM and then combining this with the camera feed. Although this would solve the perspective issue, it would also increase latency and add the complication of interfacing with SDRAM. Finally it was realised one could invert the transformation matrix and then apply it in a streaming way, as it would map the current position in the image to a flat image (the blocks). This is the approach that was chosen for rendering.

Also one had to consider how to approach marker recognition. As well as having coloured markers, markers with a black and white pattern were considered. However implementing simple, streaming code to recognise the pattern with different scaling and skew was deemed too difficult, so the coloured squares approach was chosen.

2.5 Planning and Time Management

After the change in direction of the project at the beginning of term, the team quickly began applying the aspects of the project that had been worked on to the new goal. As can be seen from the interim report before Easter, already significant progress had been made with regards to marker recognition and the VR/AR headset itself. In the Gantt chart in the aforementioned report the group chose to focus more heavily on implementing stable marker recognition before moving on to the digital logic prototyping. This meant that when it was decided to move away from digital logic solving there was no delay as all the time had been spent developing a system which could be used for almost any AR situation.

This approach also meant it was easy develop even more applications for the headset as the foundations were already in place. As a result of this, in the final few days of the project, it was decided to try and implement external HDMI input being overlaid onto the markers. This was done in less than a day (much faster than expected!) and is discussed further in Chapter 10.
Figure 2.2: Gantt Chart showing time management
Chapter 3

Elastic C Design

3.1 The Compiler

Due to limitations of the Catapult-C HLS tool which was provided, it was previously decided to find an alternative method for implementing the project. The first plan was to write the whole project in VHDL, however some tasks such as pipelining would be very tedious. As a result a new tool was created for converting a C-like language to a single-cycle, pipelined VHDL implementation, called Elastic C. This was also extended to generate state-machine based multicycle implementations, however this was not used heavily in the project.

All of the streaming processing in this project was implemented using Elastic C.

3.2 Features

Elastic C has a number of features to make it ideal for simple, low-latency streaming image processing systems, in particular:

- Automatic timing analysis and pipeline register insertion

- `stream` and `stream2d` types for creating moving windows (with automatic line buffer generation), ideal for numerous streaming image processing tasks

- ROM support, including a Python script to convert an image to Elastic C, for rendering images

- A library containing numerous useful image-related functions, such as convolution (including many common kernels) and greyscale conversion
It also supports a number of standard C constructs, in some cases with extensions to simplify common tasks:

- Data structures and arrays, automatically packed to a single `std_logic_vector` in inputs and outputs
- For loops (must be compile-time analysable in single-cycle blocks), and while loops in multicycle blocks only
- Functions, including support for an auto type specifier for creating generic functions
- Static variables, which keep their values between cycles in single-cycle blocks

The timing model used for pipeline insertion is a fairly simple model, reverse engineered by creating basic functions in VHDL and running TimeQuest timing analysis on them. Nonetheless, it generally produces implementations that pass timing analysis. The clock frequency is specified in the block header, and used to determine where pipeline stages are needed. This avoids the need for extra configuration or constraint setup steps.

Elastic C allows a very fast development cycle, with all the blocks used in the project compiling in under a second.

### 3.3 Elastic C Syntax

Elastic C’s syntax is heavily inspired by standard C, to reduce the learning curve. It has a few additions and tweaks to make it more suitable for developing hardware blocks.

Blocks are specified first using a block header, in the form of `block name(inputs) => (outputs)`. The block keyword can alternatively be `multicycle_block` to create a multicycle block rather than a single-cycle pipeline block.

The input list contains both a list of inputs (type followed by name) and optionally some special inputs. `clock<rising, 84e6>` is a typical clock specifier, creating an 84MHz rising edge clock. `enable` can be used to create an enable input for streams and static variables. Placing the `register` keyword before the input or output type creates a registered input or output.

Elastic C supports the creation of arbitrary length integers using the `unsigned<size>` and `signed<size>` types. In addition `int` specifies a 32-bit signed integer and `bool` specifies a 1-bit unsigned integer.

In addition to these basic types, arrays and data structures can also be created. These are both packed into a single `std_logic_vector` in input and output lists, with the lowest array index being the least significant bits, and the first structure element being the most significant bits.
stream<basetype, length> can also be used to create a simple shift register (using the << operator to insert values). stream2d<basetype, width, height, linewidth> creates a 2D moving window, used for example in convolution.

Finally rom<basetype, length> defines a ROM. To interface with an external ROM, an input of ROM type is used which will automatically generate an address output and data input. To instantiate a ROM with a set of values, create a variable of ROM type inside the block and use the equals operator to initialise it with a constant array (created separately outside the block, a tool is available to convert images to a constant array inside a header file) using the equals operator in the variable declaration.

An example demonstrating a range of Elastic C features is Canny edge detection, included in appendix A.1.1.

### 3.4 Principles of Operation

When compiling a single-cycle pipelined block, Elastic C converts the entire block into a series of evaluation trees. An evaluation tree is created for each output; as well as inputs and ‘write enables’ for static variables, streams and ROM address inputs. For loops are analysed at compile time and therefore must run for a constant amount of time. If statements are converted to a three-input conditional tree item for each variable changed inside the statement.

First some simple optimisations such as constant folding, conversion of division by constant to multiplication by the inverse, and balancing are performed on all of the evaluation trees.

Then a simple timing analysis is performed, to work out the maximum propagation delay from the inputs to all of the tree items. After this pipeline registers are inserted whenever the delay exceeds 90% of one clock period, and delays for tree items are recalculated whenever pipeline registers are inserted (which are considered to reset the delay to zero). Extra registers are also inserted to ensure that the latencies at the input of any tree item are equal, also taking into account the extra 1 cycle latency in accessing ROMs.

The percentage of the clock period after which a pipeline register is inserted (by default 90% for frequencies below 100MHz, and 80% above 100MHz), and the amount of slack to subtract from this figure (by default 0ns), are adjustable using two command line parameters, --budget and --slack respectively. This allows the user to balance timing performance with latency and resource utilisation.

Finally the evaluation trees are converted into a single VHDL entity, with special code to instantiate megafunctions for line buffers and non-constant division; and synthesise ROMs.

Multicycle blocks are converted into a series of assignments and code structures (such as if statements and loops), which are then analysed to see what happens at the same time. Memory
accesses are converted to a series of assignments and ‘waits’. From this, one or more finite state machines can be built. At the moment the code this generates is not highly optimised, and could be developed much further.

An illustration of how automatic pipelining would work in a simple example is shown below. First of all timing analysis works out the propagation delay at each point, this is assuming a multiplier has a propagation delay of 15ns and an adder a delay of 10ns (these are not actual figures). The output of this is in figure 3.1a.

![Diagram of timing analysis]

(a) The result of timing analysis

![Diagram after pipeline register insertion]

(b) The result of pipeline register insertion

Figure 3.1: Application of the pipelining algorithm to a simple example

After the timing analysis, Elastic C inserts pipeline registers to ensure the maximum propagation delay at any point does not exceed 90% of the clock period (this example assumes a 50MHz clock and therefore 20ns period), and that the pipeline latency at the input to all operations is the same. The system after pipeline register insertion is shown in figure 3.1b.

Please note that this example has been simplified to not take into account setup time or the propagation delay of flip flops, which the actual code takes into account.

### 3.5 Lib Elastic

An Elastic C Library has also been created in order to aid the user with image processing applications. Tasks which would normally be very complex otherwise have now been simplified down to simple functions such as `convolute_3x3()` and a number of included kernels such as...
sobel_horizontal_3x3 or gaussian_5x5. This means that relatively complex applications can be created very quickly by just applying these functions to input streams.

Simple mathematical functions have also been created to reduce code complexity and increase readability such as: abs(), clamp(), max(). This part of the library although being somewhat less impressive is far more useful to the average developer as the functions are needed in everyday programming tasks unlike the more niche image processing/convolution functions in the rest of the library.

It also contains a number of structures for common pixel formats, such as RGB666 as used in the project (with R being the MSB). These are arranged so that they are packed into a single std_logic_vector with the correct ordering when used in an input or output list.

3.6 Atom Package

All of the team were already comfortable using Atom as their text editor of choice. This led into creating an Atom package in order to make workflow with Elastic C as productive as possible. A combination of code highlighting and auto completion was decided upon as well as rapid compilation from the command line interface (CLI). Creating this package meant that the specific keywords used in Elastic C, which would not normally be picked up using normal C highlighting, could be easily recognised by the user. Also it features completion, suggesting words that the user may be wishing to type and completing key elements that never change such as brackets, and semi colons.

The Atom package is written in Coffee Script based upon a mixture of C++ and C Packages already available on the Atom Package Manager (APM) with additional Elastic C specific components. Although not publicly available, at present, the package will be distributed on the APM alongside the Elastic executables once the team are happy with their condition.

![Elastic C syntax highlighting](image)

![C syntax highlighting](image)

Figure 3.2: Package differences
Chapter 4

PCB Design

4.1 Version 1.0

As described in the interim report, version 1.0 of the PCB consisted of:

- An Altera EP3C16 FPGA, the same silicon as on the DE0 (but in a different package to make PCB layout and assembly easier)
- Two OV5640 cameras, which can run at 640x480x90fps[1], and tested up to 134fps
- A 4.8” LMS480JC01 LCD, with a resolution of 1024x600, and a maximum specified refresh rate of 80Hz[2], although it has successfully been run at up to 150Hz
- 512Mbit QSPI flash, 64Mbit SDRAM, and 16MBit asynchronous SRAM
- 4 buttons for the user interface

All major functions worked (however none of the memory devices were needed), although there were some reliability issues and some bodges were needed to get it working. This was because the crystal footprint was incorrect, and it was also discovered that a pulldown on the nCE pin of the FPGA was needed. In addition one of the LCD pins on the FPGA was not soldered successfully and had to be rerouted.

All PCB design was carried out in KiCad, which the team had experience in using and it had no limits that would affect the design. As an open source tool, this will also allow anyone to access and modify the designs without having to purchase a license, if it is decided to release the designs as open source.
4.2 Version 1.1

Although Version 1.0 was for the most part working, it was decided it would be best to have a more reliable PCB without so many bodges, and with the addition of an HDMI input, for more advanced AR systems where the content is generated on a computer, as discussed in Chapter 10.

As a result, the existing designs were modified to fix the aforementioned problems and add HDMI (this was added in place of the asynchronous SRAM, which did not prove to be useful). As much of the design was kept the same as possible, to reduce the chance of creating new issues and ensure all the existing drivers can be used as-is.

The limited FPGA resources mean HDMI input is not inside the FPGA, instead using the ADV7611 ASIC to convert the HDMI input to parallel pixel data.

In order to avoid the problems described above with soldering the FPGA, and for an overall more professional and reliable board, Version 1.1 was professionally assembled by Elecrow, the company who had fabricated the previous boards. It is worth noting that due to unclear markings on the board, the crystal oscillator was placed in the wrong orientation on Version 1.1. A replacement oscillator was manually soldered to the spare footprint.

Photos of the final assembled V1.0 and V1.1 boards are shown below.

![V1.0 Assembled PCB](image1.png)   ![V1.1 Assembled PCB](image2.png)

Figure 4.1: Pictures of assembled PCBs
Chapter 5

Driver Development

As part of the custom hardware platform that was created, VHDL drivers were written for all of the hardware components. This was done over Easter, on the Version 1.0 board, however the two boards are sufficiently similar the same drivers can be used for Version 1.1 with only changes to the pin assignments and PLL configuration necessary.

Getting high quality images out of the camera proved to be fairly difficult as OmniVision, the manufacturer, did not have the resources to support students so all development was done with the limited publicly available documentation[1][3], and a certain amount of trial and error. At the time of the previous report, raw sensor data (requiring debayering in the FPGA) had been successfully obtained from the camera, at up to 110fps. Although this worked reasonably well in good lighting conditions, a pattern of vertical lines was very visible in reduced light and the colours often ended up fairly desaturated.

It appeared that having the camera perform more processing on the image would be the best way forward, as this would be tweaked for the exact parameters of the camera. This is what proved problematic to get working - for a long time enabling the on-camera debayering and processing would result in a greyscale image only. The problem turned out, after a significant amount of trial and error, to be related to the way that vertical flip (needed due to the nature of the sensor and optics) was configured.

Once enabled, the fixed line pattern was no longer visible and the colour quality was significantly improved. However, running the camera above its maximum clock caused the camera to skip pixels producing a ‘glitchy’ image. The initial drivers for AR ran the camera at its maximum official frame rate, 90fps, however this is only possible with a vertical period of 496 lines (adding padding is possible but reduces the frame rate). As a frame buffer was not desired, and as the camera needed to match the display, some lines are duplicated (entirely line buffer based) which stretches the image slightly, however, this is a reasonable compromise for the increased frame rate without the image quality issues of doing debayering using a generic algorithm inside the FPGA.
It was later discovered that running the camera overclocked and without padding (so a stretched image) produced only a very minimal glitch (one column only) that was not visible in normal use, allowing a frame rate of 134fps. This was used in the final design.

Once the AR Minecraft demo had been completed, first a simple HDMI test design was created which passed pixels straight from the HDMI receiver to the LCD enabling PC games to be played in VR. Some debugging was needed as it emerged there was a bug in the I²C driver, that did not affect the camera, but did affect the HDMI receiver. Once this was working, the drivers were modified so that the cameras and HDMI could coexist, allowing AR applications to be developed. This is discussed in detail in [10].

The final drivers are split into several blocks:

1. A low-speed camera control driver, which handles the power on reset, and reads commands (register addresses and values) from a ROM and outputs them to the I²C driver for each camera

2. A low-speed HDMI Rx control driver, similar to the camera one

3. Two versions of simple I²C interface, one duplicated for the two cameras and one for the HDMI receiver IC

4. A high speed camera input driver, which handles the line buffer interface and combines the two 8-bit values from the camera per pixel into a single 16-bit RGB565 pixel value.

5. An HDMI input driver, which subsamples the HDMI input pixels to 128x64 and writes this into a framebuffer

6. Two half-line buffers, one for each camera

7. A display driver, which reads pixels from the half-line buffers for each camera, combines the images from the two cameras, and generates the control signals needed for the LCD

All of the image processing blocks are located between the display driver and LCD, as this negates the need for them to be duplicated for both cameras as they can work on the combined stream.

The internal PLLs are used to derive the various clocks needed for different system components: from the 50MHz input clock, an 126MHz display pixel clock, 36MHz camera input clock, 63MHz camera pixel clock and 100kHz I²C clock are all derived (these timings are for the camera overclocked to 134fps).

There is also a Nios II/e processor running on the FPGA, which carries out the less time-critical processing, in particular building the perspective transform matrices (described in detail below),
and handling block placement and inventory. The Nios was chosen as it is very resource-efficient and also has hardware floating point to speed up the matrix creation algorithm and reduce code size (memory being an important constraint.)
Chapter 6

Headset Model Design

6.1 Ergonomics and Anthropometrics

Whilst designing the physical headset itself, consideration was given to the shape and size of the human head in order to create a headset which was comfortable for the user to wear for long periods should an AR Minecraft addiction develop. In order to do this, some basic human head shapes were drawn and anthropometric data concerning the 5 - 95 percentile was gathered. Aspects to be considered with regard to the human head were as follows:

- Distance between eyes (Biectocanthus, Bipupil, Bientocanthurus)
- Nose width (Bialar)
- Nose length (Sellion-Subnasian)
- Focal Length of eye
- Whether the user wears glasses
- Forehead width (Biorbital)
- Curvature of the forehead
- Cheek bone height and width (Bizygomatic)
- Size of cranium for elastic strap (Head Length)

The weight of the headset was another concern that had to be addressed, for it needed to be comfortable for long periods of use. For this reason it was decided to not include a battery in the headset as it both reduces the mass of the item and the likelihood of it exploding in the
user’s face. It was concluded that these were key concerns. Comfort on the face would also be an issue so it was decided that it would be sensible to have foam padding on any part of the headset that made contact with the users face.

![Figure 6.1: Original sketches of male head, for measurements according to Table 6.1](image)

<table>
<thead>
<tr>
<th>Measurement Number</th>
<th>Measurement Name</th>
<th>Female Percentiles (mm)</th>
<th>Male Percentiles (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Head Length</td>
<td>175.01 198.12</td>
<td>187.96 213.11</td>
</tr>
<tr>
<td>4</td>
<td>Biorbital Breadth</td>
<td>105.92 121.92</td>
<td>107.95 128.02</td>
</tr>
<tr>
<td>5</td>
<td>Biectocanthus Breadth</td>
<td>82.04 97.03</td>
<td>84.07 101.09</td>
</tr>
<tr>
<td>6</td>
<td>Bipupil Breadth</td>
<td>54.10 64.01</td>
<td>56.01 68.00</td>
</tr>
<tr>
<td>7</td>
<td>Bientocanthus Breadth</td>
<td>25.91 35.05</td>
<td>27.94 37.08</td>
</tr>
<tr>
<td>9</td>
<td>Bialar Breadth</td>
<td>28.96 39.88</td>
<td>32.00 42.93</td>
</tr>
<tr>
<td>12</td>
<td>Bizygomatic Breadth</td>
<td>120.94 139.95</td>
<td>130.05 150.11</td>
</tr>
<tr>
<td>17</td>
<td>Sellion-Subnasian Length</td>
<td>43.94 55.88</td>
<td>46.99 60.96</td>
</tr>
</tbody>
</table>

Table 6.1: Anthropometric data from J.W. Young’s Anthropometric Report[4] corresponding to Figure 6.1

After examining all the anthropometric data, modelling of the headset began. Two features in particular required the most adjustability: the strap holding the headset to the user’s head and the lens mounts. In order to account for this, the strap was made out of elastic and grooves were located in the lens mountings so that the user could slide the lenses to the required distance apart.
6.2 Optics

The choice of lenses was largely determined by what was on the market. Custom optics are out of the budget of this project so the decision was largely based upon what was already available. Most lenses of suitable price and size had a focal length of 40mm - 50mm. It would have been nice to acquire lenses with shorter focal length in order to reduce the size of the headset. However, this may cause unwanted distortion of the image.

Trying to find a lens with a large area in focus proved to be an issue, however. After purchasing multiple different lenses and experimenting, a £10.00 perspex lens was found which appeared to work well with the headset. In the future, custom optics would be desirable in order to reduce the depth of the headset and also increase field of view.

6.3 Additive Manufacturing using Filament Deposition Modelling

In order to rapidly prototype the headset and have multiple iterations, 3D printing was the approach decided upon. Therefore, if small tweaks needed to be made down the line they could easily be implemented. The main design consisted of 4 parts:

1. Main Visor
2. Lens Holder
3. Lens Cover
4. Front Panel

The decision was made to print the headset in PLA over ABS (the two most common plastics used to print) as it is more environmentally friendly. This is due to the fact that it is biodegradable and produced from renewable resources such as corn starch or sugar cane. Using Ninjaflex for the 3d print was also suggested due to the materials flexible nature which would allow the headset to slightly adjust its shape depending on the wearers face. However, this idea was dismissed due to the extra costs of the filament and because of its environmental impact compared with say, PLA.

In the end, the headset went through 3 revisions. Version 1 consisted of just the basics needed to hold the PCB and lenses. Version 2 added holes for cables and buttons as well and honing the focal length to the millimetre. Finally version 3 had the extra features like a front cover, adjustable elastic fasteners and 3d printed buttons.
In our demo, version 2 of the headset is being used, as the exact position of lenses had been found where a double image did not occur. Also the PCB is easily accessible allowing for fixes and modifications on the fly plus easy access for JTAG.

Figure 6.2: A 3D Model of the headset (requires Adobe Acrobat Reader for 3D interactivity)
Chapter 7

Marker Recognition

7.1 Marker Design

Markers for this project consist of 2x2 coloured squares. Bright colours were chosen in order to reduce the likelihood of false positives as surroundings do not normally contain large blocks of bright colour (as discussed later). Markers consist of 4 equal squares of different colour combinations. After much experimenting with the colours, it was determined that bright red, blue and yellow were the easiest to detect and most reliable. Initial plans included using green, as this seemed the obvious choice (RGB). However, issues were encountered when detecting the markers with green on. This was mainly due to not being able to locate a printer which could print the desired colour, as green was easily and reliably identified when looking at a computer monitor. Alas this shade could not be replicated on paper. Experimentation with the surface finish of the markers also occurred and it was discovered that laser printed ones, being more reflective, were far less reliable than ink jet printed equivalents.

Lighting conditions also played a vital role in the marker detection. We found that dark, shady areas made picking up the blue and red markers challenging and that certain artificial light distorted the colour of the markers too much to be identified.

7.2 Recognition Algorithm

In order to detect the markers, a 2x2 grid is moved across the image - one pixel at a time. This is done directly on the image stream from the cameras. A check is performed to see whether there are enough pixels of either red, blue or yellow against a chosen threshold. If there are, that “block” of the 2x2 grid is assigned to its corresponding colour (or lack thereof). When the grid is over the centre of a marker it is able to determine whether the colours inside correspond
to a marker pattern. For example, detecting red-blue-blue-red indicates that a top left marker has been found.

This method is reliable, however, there can be tiny offsets as the grid does not always line up with the marker exactly. Therefore, the centre of the marker is not always perfectly found; this is also a side-effect of the fact that a check is performed to ensure each marker is only detected once for each eye. Doing the above prevents the grid technique from thinking there are multiple markers at a single position. In effect, each marker will be detected twice since the video feed consists of the combined feed from both cameras.

There are 8 boolean flags for the markers - one per marker per eye. Once the flag is set, the marker will no longer be checked for in that frame. These flags are then unset at the start of the next frame (when the x and y coordinates of the current pixel are 0). There are also 16 static variables which store the corresponding x and y coordinates of each marker for both eyes.

The Elastic C code for marker recognition is located in appendix A.1.2.
Chapter 8

Matrix Transformations

8.1 The Aim

To be able to make the augmented reality reasonably realistic the Minecraft blocks have to be transformed to the same perspective as the surface that the markers are on. This creates the effect of ‘snapping’ the image to the surface, thus augmenting the user’s reality.

8.2 Mathematical Streaming Implementation

The transformation is performed in an efficient, streaming way; this is achieved by calculating the inverse of the usual transform matrix. This moves from the current position in the image (which is effectively the transformed coordinate) to a position in the flat map. The map position can then be used to determine whether the current coordinate is in the map at all. If it is, the block type can then be determined. Its position within the block (i.e. modulo 64) determines which pixel should be accessed from the texture ROM.

Applying the matrix requires solving the system below for each pixel. Note that the final row of the matrix is constant as translation is performed separately before the transformation (this is more efficient as translation requires a lower precision).

\[
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} = \begin{bmatrix}
A & B & 0 \\
C & D & 0 \\
E & F & 1
\end{bmatrix} \times \begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]

This is implemented in two steps. Firstly:
\[
\begin{bmatrix}
  x' \\
y' \\
z'
\end{bmatrix} = \begin{bmatrix}
  A & B & 0 \\
  C & D & 0 \\
  E & F & 1
\end{bmatrix} \times \begin{bmatrix}
  x \\
y \\
1
\end{bmatrix}
\]

Then,
\[
\begin{bmatrix}
  x'' \\
y''
\end{bmatrix} = \begin{bmatrix}
  \frac{x'}{7}
  \frac{y'}{7}
\end{bmatrix}
\]

Where \(x''\) and \(y''\) are the transformed coordinates. \(x, y, x''\) and \(y''\) are implemented as signed integers; matrix values A-F are implemented as signed 3.17 bit fixed point; and \(x', y'\) and \(z'\) are implemented as signed 13.11 bit fixed point. These precisions were chosen as a reasonable compromise between accuracy and resource usage.

The Elastic C code that applies the matrix transformation, and performs all of the other rendering, is located in appendix A.1.3.

The matrix itself is generated in the Nios, using the same algorithm as the Gimp perspective tool[5]. The Nios also inverts the matrix and calculates the required translation. An exponential filter is applied to the input marker coordinates, in order to smooth out noise in the detected positions and thus improve the quality of the rendered image. However, this has some detrimental effects, discussed further in section 11.1 of the Evaluation.
Chapter 9

Minecraft Gameplay

9.1 Block Breaking

The block break button is wired as an input to the Nios core. When it is pressed, the Nios checks to see if the user is looking at a block. If they are, the entry in block RAM (a dual port RAM shared with the streaming rendering block) is set to 0 (representing air and rendered as transparent), and the inventory count for the previous block type in that location is updated.

9.2 Block Placing

Like block breaking, this is handled by the Nios core. When the user presses the block place button, the core first checks to see if the current block is empty. If it is, the Nios then verifies if the current selected block type has an inventory count greater than 0 (or creative mode is enabled), and if so, it updates the block RAM entry to the new block type and decreases the inventory count.

Two buttons are used to select which block to place, with the selected block indicated by a red outline in the inventory bar (a bar at the bottom showing the quantity and type of all blocks - see 9.3). A multicycle Elastic C block handles the block selection buttons including debouncing.

9.3 Inventory

The Nios core keeps track of an inventory for each block type, updating as described above. Alternatively the user can start the game in “Creative Mode” where an unlimited number of
all blocks are available. There is a separate dual port RAM storing 3 BCD values for each block type. These are displayed in the block picker. The Nios core converts each inventory count to BCD, suppressing leading zeros (a character code above 9 indicates blank) and setting the count to all blanks if creative mode is enabled. This reduces the amount of processing the streaming part has to do.

9.4 Maps

In order to make the game more interesting, the system supports up to 16 different maps that can be selected using different combinations of markers. These could be placed around the department, meaning players have to go round the building to find different resources. All maps are a 16x8 2D array of blocks. At present, 2 maps have been implemented.

A graphical C# application was developed that can be used to place blocks, creating the map. Alongside this is a Python script was written to convert all the maps into hex files that can be used in Quartus to initialise the map memory.

Figure 9.1: Screenshot of C# Block Picker Application
9.5 Texture Packs

As some spare Block RAM was available, the decision was made to pay tribute to the original idea of AR Doom. This was achieved by including a Doom texture pack[7] in addition to the standard textures[6]. Holding down the leftmost button during power up enables the Doom textures, otherwise the default Minecraft textures are loaded. An alternative version of the firmware was also created with a bonus texture pack for added amusement (source not included, but attempting to find the easter egg during demonstrations is encouraged.)
Chapter 10

HDMI Input

10.1 Implementation

HDMI input from a computer can be overlaid on the real, physical world. This is done using the same marker detection as the rest of the project in order to locate the surface and calculate the matrix needed for the transform. The nature of the transform means that the HDMI image pixels are not accessed sequentially, and thus fast random access memory is required. This means SDRAM cannot be used for the image, only internal block RAM, and the limited amount of block RAM available means the image has to be sub sampled to 128x64, implemented by skipping pixels, before copying it to the image RAM.

10.2 Endless Possibilities

This allows literally endless possibilities with what the user can do in augmented reality, as the power and connectivity of a computer can be taken advantage of for content generation. The wearer can have a film, the weather or even a game pinned to a wall with markers on. Using different sets of markers in order to differentiate between images one could have several augmented surfaces allowing someone to have the weather on the wall behind them, whilst watching a video on the surface in front. This could be achieved with communication over JTAG telling the computer to send different images depending on what the user is looking at.

If more memory was available, the resolution could be increased, and separate images stored for the left and right eyes, enabling 3D AR effects.

Finally, by running Doom on a PC and using the HDMI input of the headset, Dr Bouganis’s wishes for Doom in AR could be fulfilled. A device such as the Raspberry Pi could be used to make a portable setup, that the user could walk round the department with. In the future,
an ARM core could be built into the headset for generating content, either as a separate applications processor (which also has hardware 3D rendering) or integrated with the FPGA, as in Xilinx Zynq devices.
Chapter 11

Evaluation

11.1 Performance Analysis

In order to remove the jitter from the transformed image, an exponential filter is applied to the marker coordinates that are received by the Nios (see section 8.2). One of the effects observed from this, was when there was a rapid change in marker coordinates. The user sharply moving their head resulted in a noticeable delay between this movement and the transformed image being translated to the new position. The response time and accuracy of the transform is something to be improved in future iterations. However, it was deemed acceptable with respect to the specification of the project as it stands.

In addition, the marker recognition code could be improved, as in low light it can fail to reliably detect the markers; and certain colours can trigger false positives. Moving to a more complicated marker design would be a good method for removing the false positives.

11.2 Summary of Project

11.2.1 Project Photos

11.2.2 FPGA Resource Usage

The final project, with full functionality and including the drivers, uses 13,986 logic elements (91%), 416kbit of block RAM (81%) and 31 9-bit multiplier elements (28%). It could be optimised if needed by removing hardware floating point from the Nios core, and removing the extra texture pack. The older 90fps design required only 12,651 logic elements (82%), due to
the reduced number of pipeline registers required (in particular the reduced number of extra
registers needed to balance the pipeline latency at the input to each operation).

11.2.3 Timing Performance

The first designs ran the camera at its stock frame rate of 90fps and used an LCD pixel clock
of 84MHz. Running the TimeQuest Timing Analysis on this clock (which drives all of the
Elastic C blocks) gave an Fmax figure of 91.5 MHz with the “Slow 1200mV 0C Model” and
84.5MHz with the “Slow 1200mV 85C Model”. As these are worst case models, and the FPGA
is always closer to 0°C than 85°C, these figures are considered a definite success. The total
latency, including pipelining and ROM access is 14 for the rendering block (the divide in the
transform a considerable part of this), and 4 for the marker detection block.

With the frame rate increased to 134fps, and thus the pixel clock to 126MHz, the design
initially did not pass either of the slow models. Fmax figures were 110.7MHz for the 0°C
model and 102.1MHz for the 85°C model, with all of the worst-case paths being in the division.
Nonetheless, no visible glitches occurred.

Further enhancements to the Elastic C timing model were made, particularly around division
and ROM access. Thus the timing requirements were met at 126MHz, with an obtained Fmax
of 143.6MHz for the 0°C model and 132.9MHz for the 85°C model. The latency for this final
design was 26 for the rendering block and 4 for the marker detection block.

Fmax figures for the HDMI design were slightly more marginal, giving 135.9MHz for the 0°C
model and 125.3MHz for the 85°C model, however, for the reasons discussed above these do
not present any real-world problems.

11.2.4 Project Cost

The entire cost of the components for the Rev 1.1 boards, including spares for some cheaper
parts, is £100.42 in single quantities. The PCB costs £19.60 per unit in quantities of 5 (the
minimum order); the 3D printed parts cost £10.99 to manufacture as a 1kg spool of filament
had to be purchased and the lenses cost £9.00 each. This gives a total component cost of
£166.32, which even considering the extra £17.31 per unit cost for professional assembly is well
within our initial £300 budget.
11.3 Knowledge obtained

From this project the group has gained valuable experience in a wide range of relevant fields. These include the design and operation of HLS tools, PCB design, interfacing with peripherals using VHDL and the usage of image processing techniques.
Chapter 12

Conclusion

12.1 Summary of Project Achievements

During the course of this project not only has an imaginative and challenging project been created, but a framework for many future projects as well. The team learnt how to communicate and work together effectively, sharing information and files through the use of git. The group has also learnt that certain tasks have to be completed whilst creating a project, such as the tutorial book for the interim report, even though they are tedious and seem unnecessary. Time management has been a skill that has been used effectively with the project being finished with lots of time to spare, allowing the write up of the report and the creation of a poster to be completed under little time pressure.

12.2 Project Applications

With the tools developed in this project (drivers, Elastic C compiler, PCB design, Elastic C packages and the basic marker detection), what the user can do is pretty much limitless. The project’s ability to stably detect markers (and render a live transposed image on top of them) gives the user a platform to allow creative applications to be developed. HDMI input has also been included which provides data for overlaying a video on a wall. As well as this many other applications of the HDMI input are available (as discussed in \[10]\).

12.3 Future Work

There is so much which can be added to the foundations that have been laid down in this project. The most notable changes that could be made would be improving both the camera
and screen resolution, however, this would require a superior FPGA. Additions such as an inertial measurement unit (IMU) and possibly a gesture sensor mounted on the side of the headset could allow for even greater interaction with the user. With these components, one could create intuitive interfaces such as the wearer nodding after reading an instruction. Future iterations of the headset would have further adjustments on the lens position as well as a slimming down in the size and a reduction in mass of the headset. The team would love to see this project extended to augment multiple walls and surfaces with the ultimate goal being to be able to play the game doom in the EEE department. Multiple people could move around the building wearing the untethered headset as surfaces of the environment around them are replaced by Doom textures. An infrared detector could be added to the top of the headset allowing users to shoot each other with IR blasters in this augmented reality.

Another step would be to see the PCB and Elastic C language endorsed by Imperial College London. The DE0 will be 7 years old next year and as we are now at the beginning of the VR/AR era, replacing the DE0s with the VR/AR PCBs seems apt. This would allow students in future years to create inspiring virtual and augmented reality projects.
Appendix A

Code Listings

A.1 Elastic C Code

A.1.1 Canny Edge Detection

This is one of the blocks used in the demo, and shows how Elastic C can be used to implement common image processing tasks.

```c
#include "libelastic/convolution.h"
#include "libelastic/math.h"
#include "libelastic/pixel_types.h"

// arctan2, returning 0==0deg, 1==45deg, 2==90deg, 3==135deg
unsigned<2> atan2_rounded(signed<8> y, signed<8> x) {
  unsigned<2> res = 0;
  bool quad_2 = false;

  if(x < 0) {
    quad_2 = true;
    x = 0 - x;
  }

  if(y < 0) {
    quad_2 = ~quad_2;
    y = 0 - y;
  }

  if(x >= (2 * y)) {
    res = 0;
  } else {
    if(y >= (2 * x)) {
      if(quad_2) {
        res = 3;
      } else {
        res = 2;
      }
    } else {
      res = 2;
    }
  }
```

42
bool shouldSuppress(unsigned<2> [3][3] angles, unsigned<8> [3][3] mags) {
    bool is_valid;
    if(angles[1][1] == 0) {
        is_valid = ((mags[1][1] > mags[1][0]) && (mags[1][1] > mags[1][2]));
    } else {
        if(angles[1][1] == 1) {
            is_valid = ((mags[1][1] > mags[0][2]) && (mags[1][1] > mags[2][0]));
        } else {
            if(angles[1][1] == 2) {
                is_valid = ((mags[1][1] > mags[0][1]) && (mags[1][1] > mags[2][1]));
            } else {
                is_valid = ((mags[1][1] > mags[0][0]) && (mags[1][1] > mags[2][2]));
            }
        }
    }
    return !is_valid;
}

bool applyHysteresis(unsigned<2> [3][3] thresholded) {
    int i, j;
    bool valid = false;
    if(thresholded[1][1] >= 1) {
        for(i = 0; i < 3; i++) {
            for(j = 0; j < 3; j++) {
                if(thresholded[i][j] == 2) {
                    valid = true;
                }
            }
        }
    }
    return valid;
}

block cannyEdge(clock<rising, 84e6>, enable, RGB666 input) => (register RGB666 output) {
    stream2d<unsigned<6>, 5, 5, 1024> greyscale;
    greyscale << to_greyscale(input);
    stream2d<unsigned<6>, 3, 3, 1024> filtered;
    filtered << (convolute_5x5(greyscale, gaussian_5x5) / 256);

    signed<8> sobel_h = convolute_3x3(filtered, sobel_horizontal_3x3);
    signed<8> sobel_v = convolute_3x3(filtered, sobel_vertical_3x3);
    unsigned<2> theta = atan2_rounded(sobel_v, sobel_h);
    unsigned<8> mag = (abs(sobel_v) + abs(sobel_h));
A.1.2 Marker Recognition

This looks for 4 markers for each eye and outputs their coordinates to be read by the Nios core.

const int blockSize = 10;

block grid_detect(clock<rising,126e6>, enable, register unsigned<10> x,
  register unsigned<10> y, register unsigned<6>[3] input,
  register bool hsync_in, register bool vsync_in, register bool disp_in) =>
  (register unsigned<6>[3] output, bool found,
   unsigned<9> TLX1, unsigned<9> TRX1, unsigned<9> BLX1, unsigned<9> BRX1,
   unsigned<10> TLY1, unsigned<10> TRY1, unsigned<10> BLY1, unsigned<10> BRY1,
   unsigned<9> TLX2, unsigned<9> TRX2, unsigned<9> BLX2, unsigned<9> BRX2,
   unsigned<10> TLY2, unsigned<10> TRY2, unsigned<10> BLY2, unsigned<10> BRY2,
   unsigned<4> mapid, bool hsync_out, bool vsync_out, bool disp_out) {
  stream2d<unsigned<2>,3,3,1024> colour_val;
unsigned<2> COLOUR_RED = 1, COLOUR_BLUE = 2, COLOUR_YELLOW = 3, NO_COLOUR = 0;

    colour_val << COLOUR_RED;
} else {
    if((input[0] > (input[1] + input[1]/8)) && (input[0] > (input[2] + input[2]/8)) && (input[0] > 11)) {
        colour_val << COLOUR_BLUE;
    } else {
            colour_val << COLOUR_YELLOW;
        } else {
            colour_val << NO_COLOUR;
        }
    }
}

unsigned<4> red_count = 0;
unsigned<4> blue_count = 0;
unsigned<4> yellow_count = 0;

int i, j;
/* Count the number of red and blue pixels in the window*/
for(i = 0; i < 3; i++) {
    for(j = 0; j < 3; j++) {
        if(colour_val[i][j] == COLOUR_RED) {
            red_count++;
        }
        if(colour_val[i][j] == COLOUR_BLUE) {
            blue_count++;
        }
        if(colour_val[i][j] == COLOUR_YELLOW) {
            yellow_count++;
        }
    }
}

stream2d<unsigned<2>, blockSize, blockSize,1024> colour_block;

static unsigned<10> last_x, last_y;
static unsigned<10> marker_x, marker_y;

if(red_count > 6) {
    colour_block << COLOUR_RED;
} else {
    if(blue_count > 3) {
        colour_block << COLOUR_BLUE;
    } else {
        if(yellow_count > 6) {
            colour_block << COLOUR_YELLOW;
        } else {
            colour_block << NO_COLOUR;
        }
    }
}
```c
output[0] = input[0];
output[1] = input[1];
output[2] = input[2];

static unsigned<9> TLX1s, TRX1s, BLX1s, BRX1s, TLX2s, TRX2s, BLX2s, BRX2s;
static unsigned<10> TLY1s, TRY1s, BLY1s, BRY1s, TLY2s, TRY2s, BLY2s, BRY2s;
static unsigned<4> mapids;
static bool TL1found, TR1found, BL1found, BR1found, TL2found, TR2found, BL2found, BR2found;

if (x == 0 && y == 0){
    TL1found = 0;
    TR1found = 0;
    BL1found = 0;
    BR1found = 0;
    TL2found = 0;
    TR2found = 0;
    BL2found = 0;
    BR2found = 0;
}

// RBBR Top Left
if((colour_block[0][0] == COLOUR_RED) && (colour_block[0][blockSize - 1] == COLOUR_BLUE) && (colour_block[blockSize - 1][0] == COLOUR_BLUE) && (colour_block[blockSize - 1][blockSize - 1] == COLOUR_RED)) {
    output[0] = 63;
    output[1] = 63;
    output[2] = 0;
    if (x < 512){
        TL1found = 1;
        TLX1s = x;
        TLY1s = y;
    } else {
        TL2found = 1;
        TLX2s = x;
        TLY2s = y;
    }
}

// RGGR Bottom Left
if((colour_block[0][0] == COLOUR_RED) && (colour_block[0][blockSize - 1] == COLOUR_YELLOW) && (colour_block[blockSize - 1][0] == COLOUR_YELLOW) && (colour_block[blockSize - 1][blockSize - 1] == COLOUR_RED)) {
    output[0] = 0;
    output[1] = 63;
    output[2] = 63;
    if (x < 512){
        BL1found = 1;
    }
```
BLX1s = x;
BLY1s = y;
} else {
    BL2found = 1;
    BLX2s = x;
    BLY2s = y;
}

// BRRB Top Right
if ((colour_block[0][0] == COLOUR_BLUE)
    && (colour_block[0][blockSize - 1] == COLOUR_RED)
    && (colour_block[blockSize - 1][0] == COLOUR_RED)
    && (colour_block[blockSize - 1][blockSize - 1] == COLOUR_BLUE)) {
    output[0] = 63;
    output[1] = 0;
    output[2] = 63;
    mapids = 0;
    if (x < 512) {  
        TR1found = 1;
        TRX1s = x;
        TRY1s = y;
    } else {
        TR2found = 1;
        TRX2s = x;
        TRY2s = y;
    }
}

// BYYB Top Right Alt
if ((colour_block[0][0] == COLOUR_BLUE)
    && (colour_block[0][blockSize - 1] == COLOUR_YELLOW)
    && (colour_block[blockSize - 1][0] == COLOUR_YELLOW)
    && (colour_block[blockSize - 1][blockSize - 1] == COLOUR_BLUE)) {
    output[0] = 63;
    output[1] = 63;
    output[2] = 63;
    mapids = 1;
    if (x < 512) {  
        TR1found = 1;
        TRX1s = x;
        TRY1s = y;
    } else {
        TR2found = 1;
        TRX2s = x;
        TRY2s = y;
    }
}

// GRRG Bottom Right
if ((colour_block[0][0] == COLOUR_YELLOW)
    && (colour_block[0][blockSize - 1] == COLOUR_RED)
    && (colour_block[blockSize - 1][0] == COLOUR_RED)
    && (colour_block[blockSize - 1][blockSize - 1] == COLOUR_YELLOW)) {
    output[0] = 63;
    output[1] = 63;
    output[2] = 63;
    if (x < 512) {
        BR1found = 1;
    }
BRX1s = x;
BRY1s = y;
} else {
    BR2found = 1;
    BRX2s = x;
    BRY2s = y;
}

TLX1 = TLX1s;
TRX1 = TRX1s;
BLX1 = BLX1s;
BRX1 = BRX1s;
TLX2 = TLX2s;
TRX2 = TRX2s;
BLX2 = BLX2s;
BRX2 = BRX2s;
TLY1 = TLY1s;
TRY1 = TRY1s;
BLY1 = BLY1s;
BRY1 = BRY1s;
TLY2 = TLY2s;
TRY2 = TRY2s;
BLY2 = BLY2s;
BRY2 = BRY2s;

found = TL1found && TR1found && BL1found && BR1found && TL2found && TR2found && BR2found && BL2found;

// Apply the same pipeline latency as video data to sync signals
hsync_out = hsync_in;
vsync_out = vsync_in;
disp_out = disp_in;
mapid = mapids;

A.1.3 Transformation and Rendering

This handles the streaming part of the perspective transformation, and renders the block grid and block picker.

#include "libelastic/pixel_types.h"
#include "libelastic/bitmanip.h"
#include "../../../project-doom/misc/textures.bmp.h"
#include "../../../project-doom/misc/doom_textures.bmp.h"
#include "../../../project-doom/misc/map.h"
#include "../../../project-doom/misc/font_8x8.bmp.h"

block transform_variable(clock<rising, 126e6>, enable, register
unsigned<10> x,
    register unsigned<10> y,register RGB666 input,
    register signed<20>[2][3] matrix_l,
register signed<20>[2][3] matrix_r, 
register unsigned<4> map_id, 
register unsigned<8> central_block, 
register signed<11> xoff_l, register signed<11> yoff_l, 
register signed<11> xoff_r, register signed<11> yoff_r, 
register bool doom_mode, register bool grid_valid, 
register bool hsync_in, register bool vsync_in, 
register bool disp_in, register unsigned<4> 
selected_block, 
rom<unsigned<4>, 2048> map_rom, 
rom<unsigned<4>, 256> char_values) 
=> (RGB666 output, bool hsync_out, bool vsync_out, bool disp_out) {
    signed<15> x_trans, y_trans; 
    signed<11> xs = (x & 0x1FF); 
    signed<11> ys = y; 
    signed<20>[2][3] matrix; 
    int i, j; 
    // This way we reduce the number of multiplies 
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 2; j++) {
            if(x < 512) {
                matrix[i][j] = matrix_l[i][j]; 
            } else {
                matrix[i][j] = matrix_r[i][j]; 
            }
        }
    }
    // Apply the translation 
    if(x >= 512) {
        xs += xoff_r; 
        ys += yoff_r; 
    } else {
        xs += xoff_l; 
        ys += yoff_l;
    }
    // Apply the perspective transform 
    signed<24> xp = (((xs * matrix[0][0]) + ((ys * matrix[0][1])))) >> 8; 
    signed<24> yp = (((xs * matrix[1][0]) + ((ys * matrix[1][1])))) >> 8; 
    signed<24> zp = (((xs * matrix[2][0]) + ((ys * matrix[2][1])) + 0 x20000)) >> 8;
    x_trans = xp / zp; 
    y_trans = yp / zp; 
    bool is_brick = false; 
    bool is_grass = true;
    rom<unsigned<8>, 1024> font_rom = font_8x8;
unsigned<6> texture_x = (x_trans >> 2) & 0x0F;
unsigned<6> texture_y = (y_trans >> 2) & 0x0F;
unsigned<5> block_x = (x_trans >> 6) & 0x0F;
unsigned<5> block_y = (y_trans >> 6) & 0x07;
unsigned<4> map_data = map_rom[(map_id << 7) | (block_y << 4) | block_x];
rom<unsigned<18>, 4096> texture_rom = textures;
rom<unsigned<18>, 4096> texture_doom_rom = doom_textures;
unsigned<12> texture_address = (map_data << 8) | (texture_y << 4) | (texture_x);
unsigned<12> font_address = 0;
unsigned<8> char_address = 0;
bool is_text = false;
bool is_block_picker = false;
unsigned<12> rel_x = 0;
if ((y > 552) && (y < 584)){
    if ((x > 16) && (x < 496)){
        // Left eye block picker
        texture_address = ((((x - 16) >> 5) + 1) << 8) | ((((y - 552) >> 1) & 0xF) << 4) | (((x - 16) >> 1) & 0xF);
        rel_x = ((x - 16) & 0x1F);
        is_block_picker = true;
        char_address = ((((x - 16) >> 5) + 1) << 2) | (((rel_x - 4) >> 3) & 0x03);
    }
    if ((x > 528) && (x < 1008)){
        // Right eye block picker
        texture_address = ((((x - 528) >> 5) + 1) << 8) | ((((y - 552) >> 1) & 0xF) << 4) | (((x - 528) >> 1) & 0xF);
        rel_x = ((x - 528) & 0x1F);
        is_block_picker = true;
        char_address = ((((x - 528) >> 5) + 1) << 2) | (((rel_x - 4) >> 3) & 0x03);
    }
}
unsigned<18> texture_pixel, texture_pixel_doom,
texture_pixel_minecraft;
texture_pixel_doom = texture_doom_rom[texture_address];
texture_pixel_minecraft = texture_rom[texture_address];
if (doom_mode) {
texture_pixel = texture_pixel_doom;
} else {
texture_pixel = texture_pixel_minecraft;
}
unsigned<4> char_data = char_values[char_address];

// Text rendering
if (((y - 552) & 0x1F) >= 8) && (((y - 552) & 0x1F) < 16)){
    if ((rel_x >= 4) && (rel_x < 28)){
        font_address = ((char_data << 6) | ((y - 560) & 0x7) << 3) | ((rel_x - 4) & 0x7));
is_text = true;
}
}

bool font_data = font_rom[font_address];
output.R = input.R;
output.G = input.G;
output.B = input.B;

if(!is_block_picker){
// If not block picker, and in the range of the grid, and not
// an air block, render the block texture
if((x_trans >= 0) && (y_trans >= 0) && (x_trans < 1024) && (y_trans < 512) && grid_valid) {
    if(map_data > 0) {
        output.R = (texture_pixel >> 12) & 0x3F;
        output.G = (texture_pixel >> 6) & 0x3F;
        output.B = texture_pixel & 0x3F;
    }
    if(central_block == ((block_y << 4) | block_x)) {
        output.R /= 2;
        output.G /= 2;
        output.B /= 2;
    }
}
else {
    if ((selected_block == (((x - 16) >> 5) + 1))
        && (((((x - 16) >> 1) & 0xF) <= 1) || (((x - 16) >> 1) & 0xF) >= 14) ||
            (((y - 552) >> 1) & 0xF) <= 1) || (((y - 552) >> 1) & 0xF) >= 14)) {
        // The current block is the selected block and in the border region
        output.R = 63;
        output.G = 0;
        output.B = 0;
    } else {
        if ((selected_block == (((x - 528) >> 5) + 1))
            && (((((x - 528) >> 1) & 0xF) <= 1) || (((x - 528) >> 1) & 0xF) >= 14) ||
                (((y - 552) >> 1) & 0xF) <= 1) || (((y - 552) >> 1) & 0xF) >= 14)) {
            // The current block is the selected block and in the border region
            output.R = 63;
            output.G = 0;
            output.B = 0;
        } else {
            output.R = (texture_pixel >> 12) & 0x3F;
            output.G = (texture_pixel >> 6) & 0x3F;
            output.B = texture_pixel & 0x3F;
        }
    }
}

if(is_text && !font_data) {

A.1.4 Transform for HDMI Input

This is similar to above, but reads pixels from a 128x64 framebuffer (filled from the HDMI input) rather than rendering the Minecraft grid. It therefore shows the transform more clearly, without the complications of the Minecraft rendering.

```c
#include "libelastic/pixel_types.h"
#include "libelastic/bitmanip.h"

block transform_variable(clock<rising, 126e6>, enable, register unsigned<10> x,
  register unsigned<10> y, register RGB666 input,
  register signed<20>[2][3] matrix_l,
  register signed<20>[2][3] matrix_r,
  register unsigned<4> map_id,
  register unsigned<8> central_block,
  register signed<11> xoff_l, register signed<11> yoff_l,
  register signed<11> xoff_r, register signed<11> yoff_r,
  register bool doom_mode, register bool grid_valid,
  register bool hsync_in, register bool vsync_in,
  register bool disp_in, register unsigned<4> selected_block,
  rom<unsigned<18>, 8192> framebuffer)
  => (RGB666 output, bool hsync_out, bool vsync_out, bool disp_out) {
  signed<15> x_trans, y_trans;
  signed<11> xs = (x & 0x1FF);
  signed<11> ys = y;
  signed<20>[2][3] matrix;
  int i, j;
  // This way we reduce the number of multiplies
  for(i = 0; i < 3; i++) {
    for(j = 0; j < 2; j++) {
      if(x < 512) {
        matrix[i][j] = matrix_l[i][j];
      } else {
```
matrix[i][j] = matrix_r[i][j];
}

// Apply the translation
if(x >= 512) {
    xs += xoff_r;
    ys += yoff_r;
}
if(x < 512) {
    xs += xoff_l;
    ys += yoff_l;
}

// Apply the perspective transform
signed<24> xp = (((xs * matrix[0][0]) + ((ys * matrix[0][1]))) >> 8;
signed<24> yp = (((xs * matrix[1][0]) + ((ys * matrix[1][1]))) >> 8;
signed<24> zp = (((xs * matrix[2][0]) + ((ys * matrix[2][1]) + 0 x20000) >> 8;

x_trans = xp / zp;
y_trans = yp / zp;

unsigned<10> fb_x = (x_trans >> 3) & 0x7F;
unsigned<10> fb_y = (y_trans >> 3) & 0x3F;

unsigned<18> fb_pixel = framebuffer[(fb_y << 7) | fb_x];

output.R = input.R;
output.G = input.G;
output.B = input.B;

if((x_trans >= 0) && (y_trans >= 0) && (x_trans < 1024) && (y_trans < 512) && grid_valid) {
    output.R = (fb_pixel >> 12) & 0x3F;
    output.G = (fb_pixel >> 6) & 0x3F;
    output.B = fb_pixel & 0x3F;
}

// Apply the same pipeline latency as video data to sync signals
hsync_out = hsync_in;
vsync_out = vsync_in;
disp_out = disp_in;

A.1.5 Block Picker Button Debouncing

This multicycle Elastic C block debounces the block picker buttons and updates the current selected block type. This is handled outside of the Nios to enable an instant response without the complexity of setting up interrupts.
multicycle_block picker_buttons(clock<rising, 1e6>, bool btn_left,
bool btn_right) => (unsigned<4> sel_block) {
  unsigned<4> sel_block_int = 1;
  unsigned<16> holdoff = 10000; //timer for debouncing
  bool last_left_state = true;
  bool last_right_state = true;
  while (true) {
    if (holdoff >= 1000) {
      if (!btn_left && last_left_state) {
        if (sel_block_int > 1) {
          sel_block_int -= 1;
        }
        holdoff = 0;
      }
    } else {
      holdoff ++;
    }
    last_left_state = btn_left;
    last_right_state = btn_right;
    sel_block = sel_block_int;
  }
}
Bibliography


