A Hardware-based Image Perspective Correction System

Matt Hollands, Patrick Yang

6.111 - Fall 2015 - Final Report

Contents

1 Introduction

As the ability to collaborate and share digital documents becomes increasingly important both to social life and work, the ability to digitize a photograph, memo, or other document is critical. Nowadays, most people's smartphones are actually capable of this digitization process simply by pointing the phone's camera at a document. This would not be possible without the ability to detect and correct the perspective of an object, so that it appears as though one were looking at it head-on even if the photo is taken at a skewed perspective. Normally, such systems are implemented in software. For our project, we aimed to build a perspective-correction system on an FPGA, which has strict computational, memory, and timing constraints. Due to these constraints, the project carried with it a significant degree of risk.

1.1 Main Challenges

First of all, both perspective correction and image processing (for identifying edges and corners) are inherently very mathematically intensive processes. Since an FPGA has a hard limit on the number of mathematical resources available, especially multipliers, it was necessary for us to optimize our computations as much as possible to reduce the quantity of mathematical resources and ensure that we would not run over the limits.

Secondly, we chose to attempt the perspective-correction without sacrificing resolution (640×480) or color depth (30 bits from NTSC). This choice was made with the intent of replicating as close as possible the behavior of similar software-based systems, which generally do not sacrifice image fidelity. As a result, each frame takes up $307k \times 30$ bits of memory. Due to the fact that we had to compute a perspective transform, we in fact needed to store two frames in memory simultaneously. Add on the fact that image processing operations require some scratch space, and ultimately we had to manage both banks of ZBT memory on the FPGA as well as a substantial portion of the available block memory, all in tandem.

Lastly, we wanted to produce a system with decent usability, so we needed to ensure that the large number of computations being performed would not force the user to wait an egregious amount of time.

1.2 External Components

This system interfaces with the staff-provided NTSC camera as an input. The camera provides 640×480 resolution frames at 60Hz, each pixel having 30 bits of YCrCb color.

The output is to a VGA monitor with 1024×768 resolution.

1.3 Goals

Baseline goals for our system included:

- The ability to store an image from the NTSC camera into memory
- A graphical user interface allowing a user to point out the locations of the corners of an object
- The ability to perspective-transform the object and display it as if one were looking at it with direct perspective.

Ideal goals included (see below, in the Design and Dataflow section, for descriptions of these computer-vision processes):

- Automatic corner detection, consisting of the following individual components:
  - A Gaussian blur module, to reduce the noise inherent in the NTSC camera's input image
  - An edge-detection module, to find points on the source image likely to be edges of an object
  - A line-detection module, utilizing the output of the previous module to compute the locations of strong linear edges in the image
  - A corner-computing module, capable of determining the corners of a quadrilateral given four edges.

Stretch goals included:

- An upgrade to the graphical user interface, allowing the user to adjust corners with a mouse
- The ability to write the transformed image to an SD card

1.4 Results

We accomplished our baseline goals, as well as most of our ideal goals. Of the ideal goals, all the desired modules are working except for the line-detection module. The main obstacle we ran into which we could not overcome was that the side of the quadrilateral closest to the camera would register as several long lines nearly parallel to each other. We were unable to determine a satisfactory way to guarantee choosing lines that were substantially different from each other.

Nevertheless, the human interface and perspective transform work with very high performance, the transformation taking less than half a second. Furthermore, we have proven the viability of three modules we did implement from automatic corner detection, namely the Gaussian blur, edge-detection, and corner-computing.

The Verilog code can be found online in our team GitHub repository.

2 Design and Dataflow

2.1 Block Diagram

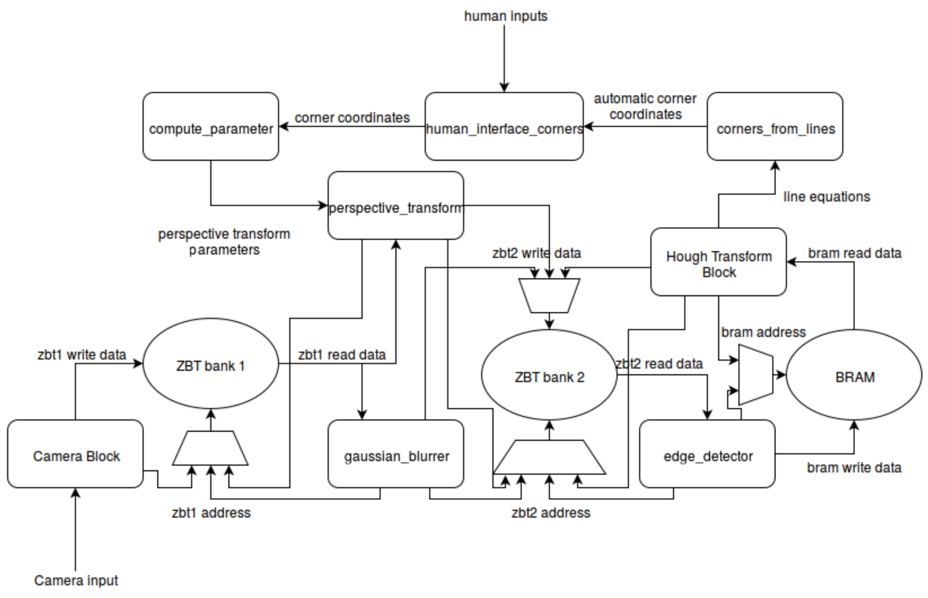

Figure 1: Block diagram illustrating main modules as well as memory and external interfaces.

2.2 Dataflow Overview

The dataflow of the system can be approximately seen as beginning in the lower left hand corner with the Camera Block, and proceeding anti-clockwise through the `gaussian_blurrer` and ending at the `pixel_transform` module. The clarity of the dataflow is somewhat obscured by the fact that the `gaussian_blurrer` module, the Hough Transform Block and the `pixel_transform` module all write to the second ZBT memory bank at different stages of the process.

The trapezoids represent signal multiplexers. For example, the multiplexer in the lower left hand corner labelled "zbt1 address" allows the Camera Block, `gaussian_blurrer` module and the `perspective_transform` module to all set the address of the first bank of ZBT memory. This is necessary so that each module can choose where data is written to or read from. Which channel is selected by the multiplexer is specified by the state of the `main_fsm` module which is not shown in this image.

2.3 Detail

In the starting state, the Camera Block is continuously loading image frames into the first bank of ZBT memory. The image is stored in the YCrCb format with 30 bits per pixel at a resolution 640x480.When the user presses the enter button on the lab kit, this prompts the `main_fsm` module to move to the next state.

In this state, the Camera Block no longer has access to the first bank of ZBT memory and so whichever image was in the bank at the time the button was pressed remains there. The `gaussian_blurrer` module now has read access to the first bank of ZBT memory and write access to the second bank of ZBT memory. It applies a Gaussian blur to the luminosity channel of the image in ZBT bank 1 and stores the result in ZBT bank 2, disregarding the two chrominance channels as they are not required for the next steps. The original image in ZBT bank 1 is not affected. When this module has completed the Gaussian blur, it asserts a DONE signal which moves the `main_fsm` module into the next state.

In this state, the `edge_detector` module has access to ZBT bank 2 instead of the `gaussian_blurrer` module. The `edge_detector` module reads the blurred image from ZBT bank 2 and performs edge detection on the image, saving the result in BRAM. The BRAM has one bit of memory per pixel which indicates if that pixel represents an edge or not. When this module has completed its process, it asserts a DONE signal to force the `main_fsm` module to move to the next state.

In this state the Hough Transform Block controls both the BRAM and the ZBT bank 2. Possible lines are calculated by reading all of the edge points from BRAM. In order to operate, the Hough Transform Module requires some temporary memory for which it uses ZBT bank 2. This is acceptable as the blurred image that is currently stored in ZBT bank 2 is no longer required. When this module has completed it will have produced four sets of lines which are described by their angle and offset from the origin. It will then assert a DONE signal to indicate to the `main_fsm` module that it should proceed to the next state.

These four lines are then passed via wires to the `corners_from_lines` module which then proceeds to calculate the coordinates of each of the four coordinates. When completed this signal asserts a DONE signal in order to tell the `main_fsm` module that it should proceed to the next state.

As the state changes, the `human_interface_corners` module is signalled to latch in the new automatically detected corner locations. These corner locations can now be modified by the user if they are incorrect by using the buttons on the lab kit. When the user is satisfied with the location of the corners, the user can then press the enter button to proceeed to the next state.

In this state, the `compute_parameters` module takes the specified corner locations and computes the parameters necessary to transform the perspective of the image. It asserts a DONE signal when complete.

In this last stage, the `pixel_transform` module has access to the first and second banks of ZBT memory. It reads the original image from the first bank of ZBT memory, performs the image transformation, and saves the resulting image into the second bank of ZBT memory. A DONE signal is asserted when complete.

The final result is a transformed image in the second bank of ZBT memory which can be displayed on the monitor.

3 Perspective Transformation

The perspective transform block consists of two modules, `compute_parameters` and `pixel_transform`. The former takes in four corners of a perspective-skewed rectangle and produces the parameters for a transformation needed to recover a perspective-corrected version. The latter iterates over the pixels in a 640 by 480 grid, applying the previously computed transformation to find the corresponding pixels from the source image.

3.1 `compute_parameters` (Patrick)

3.1.1 Motivations and Challenges

`compute_parameters` is a key module for the image transformation phase that happens after corners are identified. If we can compute the transformation that takes a direct-perspective rectangle to a skew-perspective one, then applying this transformation to individual pixels of the screen gives us the pixel from the source image that corresponds to it. The primary challenge is that the required transformation is non-linear, and furthermore, it has 9 parameters that may be very large relative to the coordinates. As a result, the module potentially requires many mathematical resources, a precious and limited commodity on any FPGA system. We have optimized the computations over previous work done by G. Ajjanagadde, S. Jain, and J. Thomas in Fall 2014, improving accuracy and greatly reducing required resources.

3.1.2 Module Details

A general perspective transformation is defined by the following equations, where the p_i are 9 parameters:

$$(x', y') = \left(\frac{p_1x + p_2y + p_3}{p_7x + p_8y + p_9}, \frac{p_4x + p_5y + p_6}{p_7x + p_8y + p_9}\right) \tag{1}$$

In our use case, what we wish to do is actually find the mapping from the direct-perspective rectangle to the skewed rectangle. In other words, (x, y) takes on the four pairs of values $(0, 0)$, $(0, 480)$, $(640, 0)$, and $(640, 480)$. Meanwhile, (x', y') takes on the corresponding four corners of the object in the source image. Since each coordinate pair gives us two equations, and we have four coordinate pairs, we have eight equations. Since the p_i can be scaled without affecting the transform, there are effectively only eight unknowns. This means that with the four corners, we have a system of eight equations and eight unknowns - exactly enough information to recover the p_i.

The reason we chose to map the direct-perspective rectangle to the skewed rectangle, and not the other way around, is that the solutions to the above equations are much nicer when (x, y) takes on four known constant values. If we were to map the skewed rectangle to the direct-perspective rectangle, we would have to then compute the inverse of the transformation, which would require significant additional resources (indeed, this was the primary source of optimization over the aforementioned previous work).

The solutions to the equations are as follows. In the below equations, x_i, y_i are the coordinates of the ith corner of the object in the source image. $c = x_4 \cdot (y_2 - y_3) + x_2 \cdot (y_3 - y_4) + x_3 \cdot (y_4 - y_2)$ is a factor that appears in several of the solutions (incidentally, it is twice the area of the triangle made up by the 2nd, 3rd, and 4th corners of the rectangle). The p_i are listed below in the order in which the module computes them, as some p_i are computed based on others.

$$p_3 = 1920 \cdot c \cdot x_1$$
$$p_6 = 1920 \cdot c \cdot y_1$$
$$p_9 = 1920 \cdot c$$
$$p_7 = 3 \cdot ((x_1 - x_4) \cdot (y_2 - y_3) + (x_3 - x_2) \cdot (y_1 - y_4))$$
$$p_8 = 4 \cdot ((x_1 - x_2) \cdot (y_3 - y_4) + (x_4 - x_3) \cdot (y_1 - y_2))$$
$$p_1 = p_7 \cdot x_4 - 3 \cdot (x_1 - x_4) \cdot c$$
$$p_2 = p_8 \cdot x_2 - 4 \cdot (x_1 - x_2) \cdot c$$
$$p_4 = p_7 \cdot y_4 - 3 \cdot (y_1 - y_4) \cdot c$$
$$p_5 = p_8 \cdot y_2 - 4 \cdot (y_1 - y_2) \cdot c$$

This module is NOT pipelined. Pipelining would require a massive number of registers, due to the large bitwidths of the computations being performed. The largest p_i, which is p_3, can be up to 42 bits long, not to mention intermediate computations. Furthermore, pipelining was deemed to introduce unnecessary development risk, as a single incorrect pipelining stage could introduce insidious and inconsistent errors. As a result, this module is allowed to run its combinational path, with the FSM guaranteeing a generous 100 clock cycles for the propagation delay to complete. In fact, we believe the propagation delay should be less than 10 clock cycles, but due to the very high clock speed giving this module extra time to complete safely was deemed to have hardly any usability cost, since this module only needs to run once per image.

As for optimizations, many common intermediate values are computed explicitly to prevent redundant additions and subtractions. c and $1920c$ are among these common values, as are all of the direct subtractions between x_i and y_i. Also, all constant multiplications are performed using a minimal number of bit shifts and addition, to further cut down on the multiplications necessary.

The result of optimization of the computations is that this module uses only 20 of the 144 multipliers available on the FPGA, leaving plenty of leeway for the other math-intensive functionality.

3.2 `pixel_transform` (Patrick)

3.2.1 Motivations and Challenges

This module computes the actual pixel transformation, given in equation (1) above, iterating through x values from 0 to 639 and y values from 0 through 479. As there are several mathematical operations in the equation, there is again room for optimization to reduce resource usage. Furthermore, this module needs to pipe data from one ZBT memory bank to the other, owing to the fact that one ZBT bank is only large enough to hold a single frame buffer with full color and resolution. As a result, we also have to time the signals to both memories in sync, accounting for any delays required in the signals.

3.2.2 Module Details

Due to optimization, this module's implementation does not use a single multiplier. We achieved this result by making use of the fact that we iterate predictably over the x and y values. We iterate in reading order, meaning that we finish a 'row' of x values before moving on to the next y value. As a demonstration, consider the numerator of (1) above, $p_1x + p_2y + p_3$. We maintain two accumulators, one for the p_1x component and one for the $p_2y + p_3$ component.

When the start signal is first pulsed, the first accumulator is initialized to 0 and the latter is initialized to p_3. This also saves an addition per pixel, since the $+p_3$ is never explicitly computed. Then, for every new pixel, we add p_1 to the p_1x accumulator; thus, the value of the p_1x accumulator will always be p_1x, even though we never compute that multiplication. If x hits 639, we then reset the p_1x accumulator to 0 and increase the $p_2y + p_3$ accumulator by p_2, again obviating multiplication. In this way, without any extra overhead, we eliminate all multiplication from the computations of the numerator and denominator for the two divisions.

Despite these optimizations, the very wide additions required still cause the combinational paths to be rather long. As a result, the module is run on a slow

(1/4 speed) clock. However, some of its components run on the original full-speed clock; in particular, the dividers used run on the full-speed clock, and any output signals must be pulsed for only one clock cycle, so the output controllers also run on the original clock.

Lastly, we will cover the timing required to correctly perform the transformation. Once the two divisions complete, we know what x', y' coordinates in the source image (ZBT bank 0) map to the given x, y coordinates in the destination image (ZBT bank 1). No image processing is required at this stage, so we wish to directly transfer data from the appropriate address in ZBT bank 0 to ZBT bank 1. The ZBT driver provided by staff has the quirk that when reading, the read data arrives 2 cycles later than the read address, but when writing, the write data, write-enable signal, and write address must all be presented on the same cycle. As a result, the write data, write-enable, and write address signals are delayed by two cycles, and the data output from ZBT bank 0 is tied directly to the data input to ZBT bank 1. Determining the correct set of signals to delay was somewhat challenging due to the complex timing specifications.

Future work in this area would involve pipelining the divider and the additions needed to generate the numerators and denominator of the divisions. While this wouldn't produce a very perceptible increase in speed with the existing design, it would be a critical step towards using this system for any sort of real-time application.

4 Memory and IO

4.1 `human_interface_corners` (Matt)

The `human_interface_corners` module provided the ability to manually specify the four corners of the document of interest. It took input from the four directional buttons and the four numbered buttons of the labkit in order to allow the user to select a corner and move it to a desired location. On top of this, it had to allow the automatic corners to override the manual corners when a new set of automatic corners were calculated. This module was important from the early stages of the project as it was key in allowing the perspective transformation block to be tested. It was also a critical module as it meant that, in the event of incorrectly detected corners, the user could specify the location of the corners and still produce a useful result.

4.2 `corner_sprite` (Matt)

This module was an implementation of a sprite that drew a simple cross at a given location on the screen. A colour for the cross could be specified. By telling the module which pixel on the display was currently being sent to the display, the sprite could determine and return whether or not this pixel lay on the sprite and hence whether or not it should be coloured in.

4.3 `vram_display` and `bram_display` (Matt)

The `vram_display` and `bram_display` modules take as input the coordinates of the current pixel being drawn to screen. From this they could then calculate the memory address of the given pixel in ZBT or BRAM memory such that the correct image would be displayed to screen.

The only difference between the `vram_display` and `bram_display` modules is that the `vram_display` module had to forecast which pixel would be drawn next in order to access the ZBT memory in time (due to the two cycle access time), whereas the `bram_display` module did not.

4.4 `main_fsm` (Patrick and Matt)

The `main_fsm` module controlled how the entire system operated. Inputs for this module were the enter button and switch[7] of the labkit as well as DONE signals from all of the image processing and perspective transform modules. This allowed states to be changed when the user pressed the enter button (e.g. to begin edge detection), or when one module finished and another one should start (e.g. after the `compute_parameters` had completed, the state should change to start the `perspective_transform` module). Output from this module were the 5 bit state number as well as START signals for all of the image processing and perspective transform modules. Overall the system uses 27 states.

5 Image Processing

5.1 `gaussian_blurrer` (Matt)

5.1.1 Motivations and Challenges

The first stage in the image processing step was to perform a Gaussian Blur on the image. This had the effect of removing spurious noise from the image, allowing the edge detection steps to find cleaner edges. Originally it was attempted to run the edge detection module directly on the unprocessed camera image, however the result was lots of ”edges” that were actually due to noise in the image.

The main challenge of this module was a result of the inherent delay in the ZBT memory. The final value for each output pixel is a linear combination the 24 input pixels bordering it and itself. As a result, at each point in time, 25 pixel values had to be held in registers. Each time the x-coordinate of the pixel being processed increased, 5 new values had to be loaded into registers from

ZBT memory, and each time the y-coordinate increased by one, 25 new values had to be loaded into registers from ZBT memory.

Another timing issue arose from the fact that this process required 25 multiplications per pixel, which could not be achieved in a single clock cycle due to the propagation delay of the multipliers available. Furthermore, the weights of each neighbouring pixel were not an integer which meant that normal integer binary representation would not be sufficient.

5.1.2 Module Details

Originally a 3x3 Gaussian kernel with a standard deviation of 1.4 was used, however it was found that a 5x5 kernel with standard deviation of 2 gave a better result due to the high level of noise in the image.

The module read data from the first bank of ZBT memory and wrote data to the second bank of ZBT memory. This was necessary because the original image in the first bank of ZBT was need for the perspective transformation step and so it could not be overwritten.

In order to overcome timing challenges due to ZBT memory access delays and multiplication propagation delays, the module consumed 26 clock cycles per pixel calculation. This resulted in approximately one multiplication per clock cycle and also gave plenty of time for the 5 ZBT memory reads required for the next pixel.

The module did not reload all 25 pixel values from memory each time the module moved to the beginning of a new row of pixels and instead used whatever values were already in memory. This meant that the first 5 pixels in each row were incorrectly calculated, however it was decided that this was insignificant as it was unlikely that the edges of the document lay in this region of the image.

The pixel weights were approximated as a fraction with denominator 1024. This meant that the calculation could executed as a multiplication by the numerator, followed by a 10 bit right shift. The inaccuracies introduced by this approximation were very small, on the order of 1%.

$$\begin{bmatrix} 0.023528 & 0.033969 & 0.038393 & 0.033969 & 0.023528 \\ 0.033969 & 0.049045 & 0.055432 & 0.049045 & 0.033969 \\ 0.038393 & 0.055432 & 0.062651 & 0.055432 & 0.038393 \\ 0.033969 & 0.049045 & 0.055432 & 0.049045 & 0.033969 \\ 0.023528 & 0.033969 & 0.038393 & 0.033969 & 0.023528 \end{bmatrix}$$
$$\approx \begin{bmatrix} \frac{24}{1024} & \frac{35}{1024} & \frac{39}{1024} & \frac{35}{1024} & \frac{24}{1024} \\ \frac{35}{1024} & \frac{50}{1024} & \frac{57}{1024} & \frac{50}{1024} & \frac{35}{1024} \\ \frac{39}{1024} & \frac{57}{1024} & \frac{64}{1024} & \frac{57}{1024} & \frac{39}{1024} \\ \frac{35}{1024} & \frac{50}{1024} & \frac{57}{1024} & \frac{50}{1024} & \frac{35}{1024} \\ \frac{24}{1024} & \frac{35}{1024} & \frac{39}{1024} & \frac{35}{1024} & \frac{24}{1024} \end{bmatrix}$$

Figure 2: The 5x5 Gaussian kernel with standard deviation 2 can be approximated as fractions with denominator 1024

5.2 `edge_detector` (Matt)

5.2.1 Motivations and Challenges

After the Gaussian Blur, it was necessary to identify pixels in the image which represented edges - these would likely indicate the edges of the document of interest and therefore could be used to find the corners of the document.

In order to achieve this, the Canny Edge Detection algorithm was used. This algorithm relies on convolving the luminosity channel of the image with the Sobel operator to calculate the luminosity gradient of each pixel in the horizontal and vertical directions.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A$$
$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * A$$
$$G = \sqrt{G_x^2 + G_y^2}$$

Figure 3: The Sobel Operator applied to 3x3 matrix A which represents the pixel of interest and 8 bordering pixels

A pixel is considered an edge if the magnitude of the gradient, G, is greater than a certain threshold.

5.2.2 Module Details

Similarities can be seen between this module and the `gaussian_blurrer` in that they both convolve an image with a two dimensional kernel. Therefore this module shares many implementation details with the `gaussian_blurrer` module. Each pixel requires 8 clock cycles for processing. During 6 of these cycles, the Sobel operator is applied to the local pixel to calculate G_x and G_y. In the next two cycles G_x^2 G_y^2 are calculated and in the final cycle the sum of the two squares is compared to a threshold to decide whether or not the pixel is an edge. Interspersed throughout these cycles are the ZBT memory reads to acquire the next three pixel values.

To store the results of these calculations, one bit of BRAM memory is used where a 1 represents that this pixel is an edge and a 0 represents that this pixel is not an edge.

5.3 `sin_lookup` and `cos_lookup` (Patrick and Matt)

These two modules were simply used as sine and cosine look-up tables. They took an input of an 8 bit unsigned integer representing an angle and would return the sine or cosine of that angle. Because these functions would return a number between -1 and +1, the result was scaled by a factor of 4096 so that the result would be a 13 bit signed number.

Due to the nature of our calculations, the look-up tables only needed to return valid results for angles that were multiples of four between 0 and 176 degrees.

5.4 Hough Transform Block

The Hough Transform Block consisted of four modules which, between them, performed the Hough Transform algorithm. These modules are:

- `hough_transform_coordinate`
- `hough_transform_calculate`
- `hough_mem`
- `hough_transform_find_highest`

5.4.1 `hough_transform_coordinate` (Matt)

The Hough Transform algorithm is split into a number of parts, and as a result it is necessary to ensure that each of these parts occur in the correct order. Therefore it was required to have a coordinating module which manipulated START and DONE signals in order to control the other three Hough Transform modules. The first step was to use the `hough_mem` module to clear a space in ZBT bank 2 memory in order to store the strength of each possible line in the image.

The next step was to go through every pixel in the image and, if it was an edge pixel (as detected by the `edge_detector` module), increment the strength of each possible line that passed through that point using the `hough_mem` module.

Finally, the last step was to go through every possible line and find the four strongest lines by finding which lines had been incremented the most number of times. This was achieved using the `hough_transform_find_highest` module.

5.4.2 `hough_transform_calculate` (Matt)

One major part of the algorithm requires calculating all possible lines through a given point. In our case, we grouped lines into buckets of similar slope, with buckets of size 4 degrees (resulting in 45 possible lines: 0 degrees, four degrees,..., 176 degrees). This module would take as input the x and y coordinate of a point. It would then calculate the parameters ρ and θ to describe each of the 45 possible lines that could pass through the pixel. These line parameters are then outputted on wires to the `hough_mem` module at a rate of one per two clock cycles.

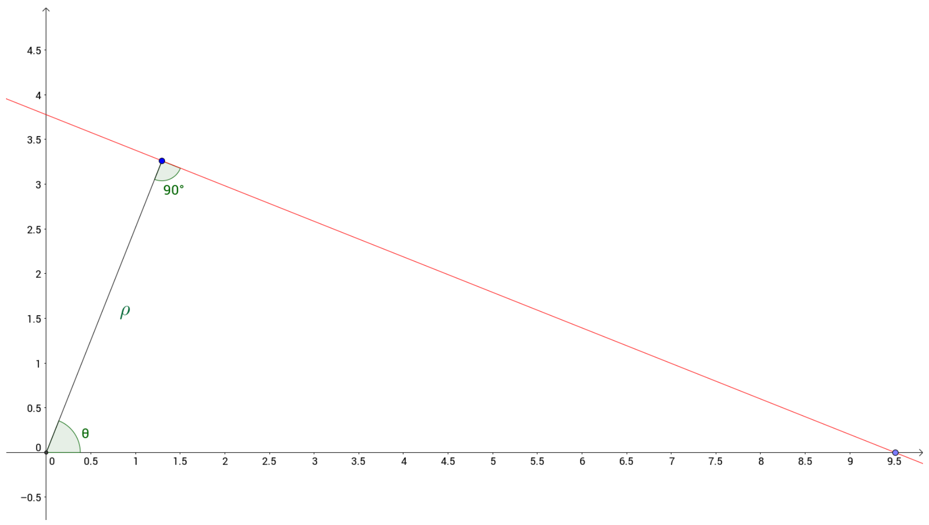

Figure 4: An illustration of the ρ, θ parametrization of a line.

5.4.3 `hough_mem` (Patrick)

This module controlled manipulation of the second bank of ZBT memory in order to keep track of the strength of each possible line. Given the parameters of a given line, it could load the strength of that line from the ZBT memory, increment the value by one and rewrite the new value to memory. This update action was pipelined, enabling 45 updates to be processed in just over 90 clock cycles. The timing turned out to be very specific and tight, and often this module juggles two actions (read and write) at different stages simultaneously.

Furthermore, given a particular start signal it was able to clear the second ZBT memory bank in order to initialize its values to 0.

5.4.4 `hough_transform_find_highest` (Matt)

This module was the final step of the Hough Transform. It passed over every possible line in memory and selected the four lines with the highest strength. Ideally, these four lines would represent the four edges of the original document.

In practice it was found however that four very similar lines were selected. This was likely due to the fact that the edge detector found multiple "edge pixels" at each point along an edge which resulted in a thick line of "edge pixels". As a result, multiple lines that were very similar but with slightly different slopes or offsets were selected.

This could potentially have been fixed by using a thinning algorithm after the edge detection in order to thin the number of edge pixels found at each point along the edges of the document and hopefully produce thinner edges, however time restrictions did not allow this to be completed in time.

5.5 `corners_from_lines` (Patrick)

5.5.1 Motivations and Challenges

This module takes four lines identified by a Hough Transform and outputs the four corners of a convex, non-self-intersecting (i.e. normal) rectangle defined by those four lines. The need for this module was recognized relatively late during our implementation process, due to the fact that often software implementations abstract out, or otherwise implicitly perform this computation.

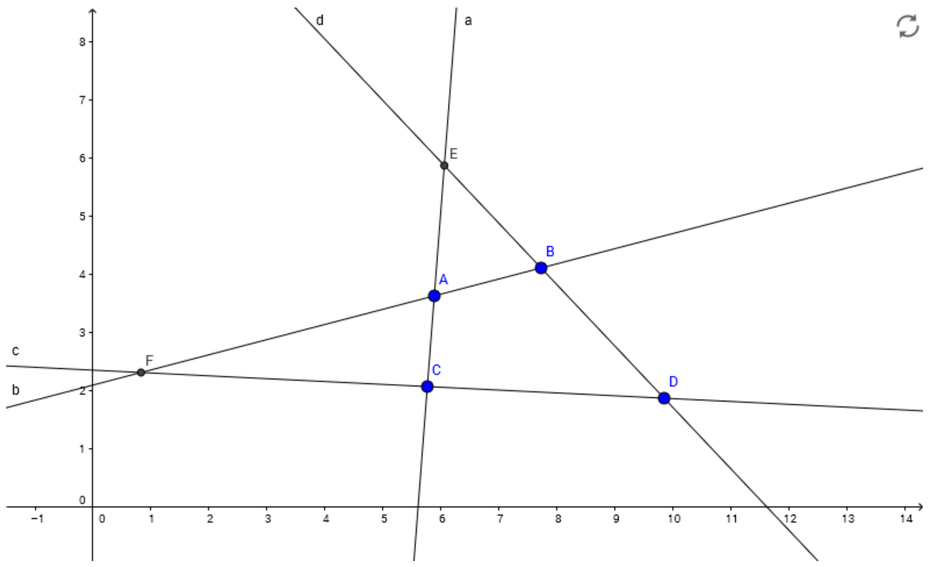

Figure 5: An illustration of the complexity behind extracting four sensible corners from four lines.

Given four lines as in the figure above, there are actually three 'quadrilaterals' which might be extracted. $ABCD$ is the desired quadrilateral, but $AFDE$ and $CFBE$ are also quadrilaterals, and there is no immediate way to distinguish between them. Thus this module not only needs to perform line intersection computations, it needs to algorithmically rule out invalid quadrilaterals.

5.5.2 Module Details

First, we will cover line intersection. The Hough Transform produces lines in Hesse Normal Form, which means that the equations of the lines are given by $r = x\cos(\theta) + y\sin(\theta)$. With two lines of this form (or more specifically, the parameters $r_0, \theta_0, r_1, \theta_1$), the x and y may be derived as follows:

$$(x, y) = \left(\frac{r_1 \sin(\theta_0) - r_0 \sin(\theta_1)}{\cos(\theta_1)\sin(\theta_0) - \sin(\theta_1)\cos(\theta_0)}, \frac{r_0 \cos(\theta_1) - r_1 \cos(\theta_0)}{\cos(\theta_1)\sin(\theta_0) - \sin(\theta_1)\cos(\theta_0)}, \right)$$

We optimize this computation by noting that the denominators are actually the same value, and that furthermore this value is equal to $\sin(\theta_0 - \theta_1)$ by the sine sum-of-angles formula. For all trigonometric computations, this module reuses the lookup tables described above. For division, this module utilizes the staff provided divider.

As there are six intersections to be computed, this module utilizes six copies of the line-intersection submodule to compute them simultaneously. This is very resource-intensive, and the choice was made for simplicity due to the lateness of this module's creation. Resource use could easily be improved by computing the six intersections sequentially.

Next, we discuss the identification of the 'correct' rectangle from the six computed intersections. The algorithm for determining whether a rectangle $ABCD$ is valid (convex and non-self-intersecting) is as follows.

Let $c(ABC)$ be the orientation of triangle ABC, namely whether points A, B, C would be clockwise around the triangle. Then, $ABCD$ is valid if $c(ABC) = c(BCD) = c(CDA) = c(DAB)$. The algorithm for computing $c(ABC)$ is as follows, and is sourced from http://algs4.cs.princeton.edu/91primitives/:

$c(ABC) = (B_x - A_x)(C_y - A_y) - (C_x - A_x)(B_y - A_y)$.

Four copies of the orientation-computing submodule comprise a submodule that checks a rectangle for validity; this validator submodule is used sequentially on the three candidate rectangles to find one that works.

6 Teamwork, Timeline, and Logistics

Initially, the work was approximately distributed between team members such that Patrick would do most of the perspective transformation work and Matt would do most of the human interface and image processing work. This was generally the case throughout the project.

The largest deviation from the original timeline was due to the image processing requiring more steps than expected. A good example of this is that we did not expect to have to perform a Gaussian blur on the image, however after seeing the high noise levels produced by the camera we found that this was absolutely necessary. Implementing this module took a reasonable amount of time.

Both team members agree that we spent approximately equal durations of time working on the project and communication was effective.

GitHub was used in order to allow simultaneous development; however it was found that as each branch was developed, naming conventions and memory handling architectures were quite different in each, resulting in all integration and merging having to be done entirely by hand. This resulted in some time penalty. In the future, we would recommend teams to set up a framework within which to build modules as one of the first actions (i.e. do 'integration' first rather than last). At the very least, having things like common top-level connection naming, for things like memory wires, would be highly beneficial.

7 Conclusion

The product fulfilled all of the baseline goals laid out at the start of the project. It provided the user with an interface with which she could take an image of a document at a skewed perspective, manually select the four corners and automatically correct the perspective of the image.

The ideal goals were very nearly completed, with most steps in the image processing chain demonstrably working correctly. The project is left in a state where, given more time, the Hough Transform could be perfected in order to detect the correct lines in the image. With this step complete, the `corners_from_lines` module could calculate the correct corner locations and then the whole process could become automated.

The stretch goals were not completed.

This project was known to be ambitious from the start and we are proud of what we have achieved. Some disappointment is found in being close to completing the ideal goals, but not quite there. However, overall the process has been an educational and enjoyable one and we would like to thank Professor Gim Hom and all of the teaching and laboratory assistants from their help in making the project a success.