2010

COMMERCIALIZATION AND OPTIMIZATION OF THE PIXEL ROUTER

Steven James Dominick
University of Kentucky, steve.dominick@gmail.com

ABSTRACT OF THESIS

COMMERCIALIZATION AND OPTIMIZATION OF THE PIXEL ROUTER

The Pixel Router was developed at the University of Kentucky with the intent of supporting multi-projector displays by combining the scalability of commercial software solutions with the flexibility of commercial hardware solutions. This custom hardware solution uses a Look Up Table for an arbitrary input to output pixel mapping, but suffers from high memory latencies due to random SDRAM accesses. In order for this device to achieve marketability, the image interpolation method needed improvement as well. The previous design used the nearest neighbor interpolation method, which produces poor looking results but requires the least amount of memory accesses. A cache was implemented to support bilinear interpolation to simultaneously increase the output frame rate and image quality. A number of software simulations were conducted to test and refine the cache design, and these results were verified by testing the implementation on hardware. The frame rate was improved by a factor of 6 versus bilinear interpolation on the previous design, and by as much as 50% versus nearest neighbor on the previous design. The Pixel Router was also certified for FCC conducted and radiated emissions compliance, and potential commercial market areas were explored.

Keywords: Pixel Router, Cache, Bilinear Interpolation, LUT, Commercialization

Steven James Dominick

12/15/2010

COMMERCIALIZATION AND OPTIMIZATION OF THE PIXEL ROUTER

By

Steven James Dominick

Dr. Bruce Walcott
Co-Director of Thesis

Dr. Ruigang Yang
Co-Director of Thesis

Dr. Stephen Gedney
Director of Graduate Studies

12/15/2010

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

THESIS

Steven James Dominick

The Graduate School

University of Kentucky

2010

COMMERCIALIZATION AND OPTIMIZATION OF THE PIXEL ROUTER

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Electrical Engineering in the
College of Engineering
at the University of Kentucky

By

Steven James Dominick

Lexington, Kentucky

Co-Directors: Dr. Bruce Walcott, Professor of Electrical Engineering
and Dr. Ruigang Yang, Professor of Computer Science

Lexington, KY
2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FILES

Section 1: Introduction

Multi-Projector Displays

The utilization of multiple projectors to create a single seamless, uniform display is quickly becoming a mature technology. Television newscast sets, command and control centers, large conference halls, houses of worship, and many other markets take advantage of this technology to enhance the presentation of visual information. By overlapping and "edge-blending" two or more video projectors, one can create a resulting image with higher resolution and brightness and less throw distance between the projection and the projector lens.

Software Blended Displays

There are two primary approaches to creating edge blended displays that are currently available in the market. The first approach is a software-based process that runs on commodity hardware. Typically, this process involves camera feedback to automatically align and blend the projectors to create blending and warping masks that result in a perfectly aligned and blended seamless display. This process is the same for any number of projectors in any configuration, thus making it an extremely scalable method. The calibration procedure in software can be performed in a few minutes or seconds, even for large arrays of projectors. In 2005, Brown et al presented a survey of available techniques for camera-based projector calibration, comparing capabilities and computational requirements for each method [1]. In this survey, the two primary metrics used to compare calibration techniques consisted of geometric registration and photometric correction. Calibration techniques can compute the geometric registration on either planar surfaces or arbitrary surfaces, using one or several cameras. On an arbitrary (non-planar) surface, the geometric warping can be computed for either a stationary viewer with a single ideal head position or for a moving viewer where the head position is tracked. This second method, known as 3D global registration, allows for the creation of immersive displays capable of displaying 3D rendered content that correctly accounts for a user's viewpoint. The other important component to software blended displays is the photometric correction of the display. In the paper, Brown discusses variables to which photometric correction can be applied: intra-projector variance, inter-projector variance, and overlap variance. Intra-projector variance is a result of non-uniformities within the display of a single projector which can result from properties of the projector lamp or the display surface itself. Inter-projector variance is the difference in luminance between separate projectors. Even projectors of the same make and model can have differences in color and intensity due to manufacturing tolerances and uneven aging of projector lamps and other components in the device. Both intra-projector and inter-projector variance are small, however, compared to variance in the overlap regions between projectors. In the software blending methods discussed, overlap blending is the primary focus for the photometric correction. These methods use a linear or cosine ramp function in the overlap regions of the projectors to

produce smooth transitions between the display area of one projector to another. Brown gives an example of linear ramping by considering two overlapping projectors, P_1 and P_2, whose contributions at a location *x* on the screen on the display surface are $P_1(x)$ and $P_2(x)$, respectively. The blended intensity at *x* is calculated by the equation below, where $\alpha_1+\alpha_2 = 1$.

$$I(x) = \alpha_1(x)P_1(x) + \alpha_2(x)P_2(x)$$

The alpha weights α_1 and α_2 are calculated based on the distance of *x* from the boundaries of the overlap region. For a linear ramp function, α_1 and α_2 are calculated as follows, where d1 and d2 are the distances to the edges of the overlap region.

$$\alpha_1(x) = \frac{d1}{d1 + d2};\ \alpha_2(x) = \frac{d2}{d1 + d2}$$

Brown et al also give an example of the resulting blending masks from 4 overlapping projectors using this linear ramp function blending technique. In that image, shown below in Figure 1 [1], the left most images are the alpha blending masks for these projectors, the middle image is the unblended but geometrically corrected image, and the right image is the geometrically and photometrically corrected image using these alpha masks.

Figure 1 – Alpha blending mask example [1]

Software blended displays are not without their disadvantages, however. The output images to the projectors in the display are warped and blended through commodity graphics hardware in a PC. However this hardware does not natively support blending and warping functions in the low-level drivers, thus an application must be written at the Operating System level which applies the geometric and photometric correction to any content that is desired to be displayed. Generally, applications written for an OS are not designed to support blended multi-projector output, and thus source code modifications are required to enable this feature. Most applications use proprietary source code that is not possible to modify as an end user, thus content is difficult to generate for software blended displays. This limitation reduces the number of applications that software blended displays can be useful for to scenarios where the

content source code can be accessed either through open source modification or active cooperation with the company that produces the software. Thus, software blended displays tend to only be used in high-end simulation and visualization applications that use specialized image generation software that can be easily modified.

Hardware Blended Displays

Another approach to creating multi-projector displays is to use customized hardware that acts as an image processing "pass-through" device. These devices have one or more video inputs and at least two video outputs, and apply basic warping and blending techniques to achieve geometric and photometric uniformity on a display. Unlike software blended displays which use camera feedback to automatically compute the warping and blending masks, these devices rely on a manual calibration process for image alignment and correction. To compute this correction, projector is connected to each of the device's video outputs and some alignment tool, such as a displayed grid pattern, is used to assist the user in manual overlap correction. Once the display has been properly aligned and blended, the inputs on the device allow the user to treat the entire display as a single monitor. Any compatible video input plugged in to the blending device will be shown across all projectors in a seamless, uniform manner. The major disadvantage of systems such as these is setup time. Consider a projector array with *M* projectors in the horizontal direction and *N* projectors in the vertical direction. Aligning a *1xN* or *Mx1* array is not a terribly complex task using these systems, but alignment difficulty grows dramatically with an *MxN* system, where *M* and *N* are both greater than 1. In a *1xN* or *Mx1* array, no projector overlaps with more than 2 other projectors. However in an *MxN* array, a projector may overlap with many other projectors, creating more variables and adjustment parameters than can be accounted for in a short period of time. As the number of projectors is increased, the complexity of setup increases exponentially. This presents a significant disadvantage as compared to software blended displays, whose calibration time is a linear function to the number of projectors in the display. Thus, hardware-based blended displays typically are more useful in long term, stationary environments than in situations where there is need for frequent realignments.

The Pixel Router

The Pixel Router is a custom hardware device developed at the University of Kentucky with the intention of bridging the gap between software and hardware edge blending. The device, which has 4 HDMI inputs and 4 HDMI outputs, uses look-up tables (LUTs) generated by a software calibration process to warp and blend up to 4 input images across the outputs. These LUTs represent alpha masks such as the one shown in **Error! Reference source not found.**Figure 1, and apply photometric alpha blending to correct for overlap variations. Additionally, the LUTs can map any given input pixel to any given output pixel, allowing for the implementation of the geometric correction calculated by the calibration software [2]. This device takes advantage of the scalability and flexibility of the software calibration procedure while maintaining the content agnostic properties

of hardware based solutions. This thesis will focus on efforts to improve the performance of the Pixel Router and potential opportunities for commercialization. The primary area of performance improvement will be to improve the appearance of rotated, warped, and scaled images while maintaining an acceptable frame rate. This will be achieved through the implementation of bilinear interpolation and the design of an efficient cache that improves the performance of this memory-intensive operation. Additionally, packaging the device and securing proper FCC certification will be explored.

Section 2: Previous Work

Pixel Router Design

The Pixel Router is the result of a number of years of work and research at the University of Kentucky (and the University of North Carolina Chapel Hill?), and has undergone several design iterations. The current version is implemented on a Xilinx Virtex-4 FPGA, part number XC4VLX40-FF1148. Four 32Meg x 32-bit DDR SDRAM memory banks are used to store LUT data as well as input and output video frame buffer data. These memory banks are each made up of two Micron MT4632M16P-5B devices and operate at 133MHz for a data rate of 266MHz. The HDMI interface uses 4 Analog Devices AD9398 receivers on the input side and 4 Analog Devices AD9889B transceivers for video transmission [PR Spec]. This allows the Pixel Router to receive and transmit 4 independent HDMI channels each capable of handling video at up to 1080p resolution (1920x1080 pixels).

The Pixel Router is designed with a Look Up Table architecture that allows any input pixel to be mapped to any output pixel, with the pixel color value multiplied by an alpha to allow for image intensity blending. The Look Up Table uses an inverse mapping function where the input pixel address required is determined by the output pixel location. This is because an input pixel can be assigned to multiple output pixel locations, but not vice versa. A diagram showing this inverse mapping concept is shown below. In this diagram, the [X, Y] coordinate system represents the output pixel space and the [U,V] coordinate system represents the input pixel space.

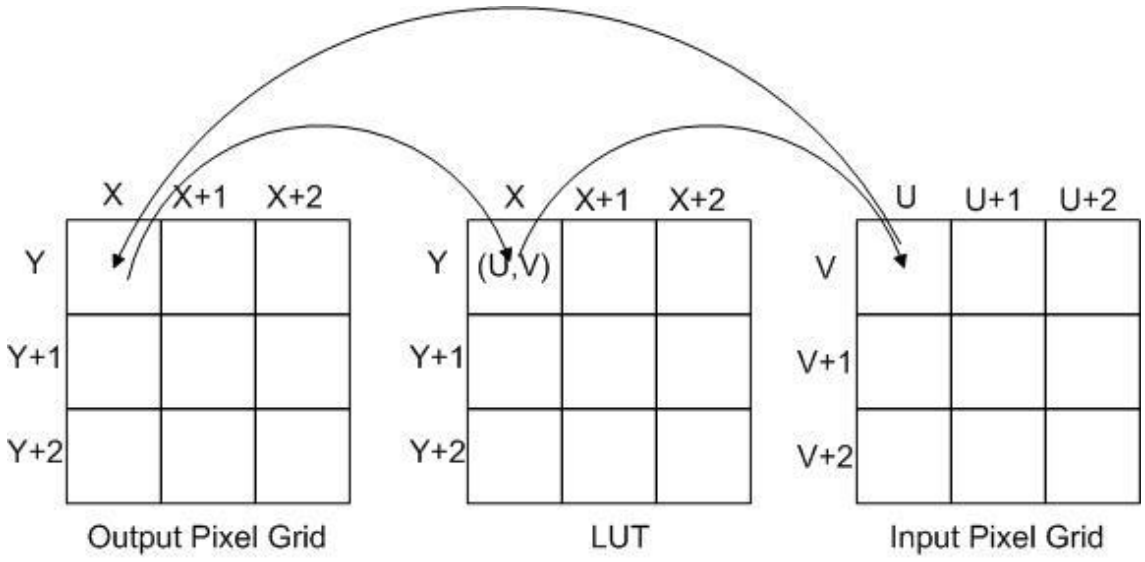

Figure 2 - Inverse Mapping

These features make the Pixel Router a flexible device capable of performing arbitrary image warping in a variety of applications. The intent of the device is act as a stand-

alone hardware platform to perform the image blending and warping required for multi-projector displays, keeping a computer out of the loop.

Cache with Blocks

Previous work has been done by Vijai Raghunathan [3] to design a cache for the Pixel Router that improves performance and allows for bilinear interpolation to be implemented. In this project, an optimal cache was designed that allowed the Pixel Router to operate at reasonable frame rates independent of the amount of image rotation. This cache system used the concept of "Memory Blocks", which divides the memory space into blocks and effectively rearranges the access pattern to minimize memory latency. In this design, it was determined that the optimal block size was 64x64 pixels, and the optimal cache size was 64x32 pixels. Below is an image comparing the performance of this cache system to direct SDRAM access [3].

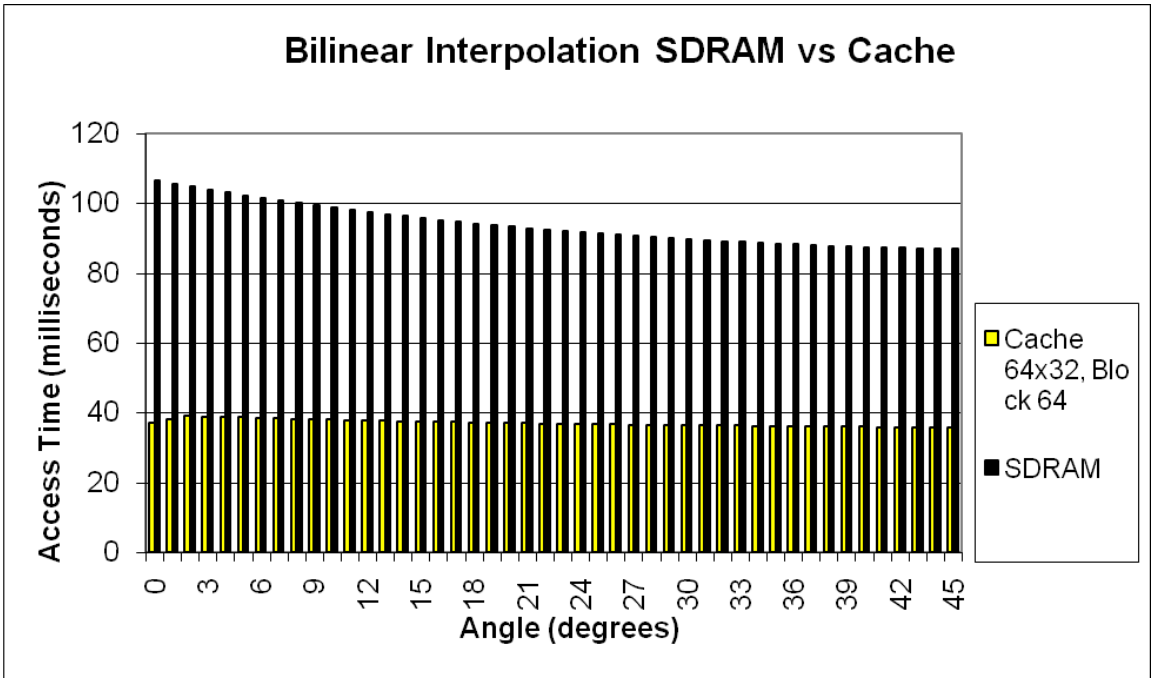

Figure 3 - Cache with Blocks Simulation [3]

This simulation measured the SDRAM access time for one 1024x768 pixel image frame across different amounts of image rotation, with bilinear interpolation applied to improve the image quality. The simulations suggest that this cache design provides a significant performance improvement over direct SDRAM access. Additionally, this cache design is independent of the amount of image rotation, as the access time remains nearly flat at just under 40 milliseconds.

Another way of describing the performance of cache design that is perhaps more intuitive is in terms of output frame rate rather than access time. For these simulations,

the access time for one output frame was measured. In the Pixel Router, however, there are 4 output ports, and each one is processed sequentially rather than in parallel. The block diagram below is taken from the Pixel Router Specification [3] and shows the data path of the video output signals.

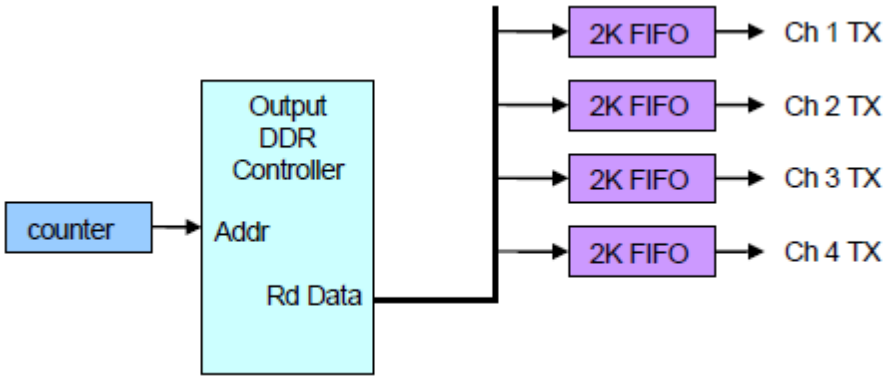

Figure 4 – Video Output Data Path [3]

The transmit FIFOs in the diagram above are written sequentially in 256 word bursts, and therefore the number of pixel values output from the Rd Data port on the Output DDR Controller is equal to 4 times the output resolution on each channel. This creates a bottleneck in processing the data, and that same bottleneck is present on the interface between the FPGA and the Input DDR Controller. Thus when considering the real frame rate on the Pixel Router, the access time presented in Raghunathan's thesis should be multiplied by 4. Parallelizing these memory interfaces could help improve bandwidth and therefore frame rate, but such work is outside of the scope of this project as the goal is to optimize the performance for the current hardware revision. The conversion for the access time in the simulations above to Pixel Router frame rate is shown below.

$$Frame\ Rate = \frac{1}{4 * Access\ Time}$$

The access time in these simulations was approximately 35 milliseconds, which corresponds to a frame rate on the Pixel Router of just over 7 frames per second. While this marks an improvement over implementing bilinear interpolation on previous design with no cache architecture, it is below the design target of 60 frames per second, and also below the nearest neighbor operation of 20 frames per second. Additionally, it was determined that implementation of this cache system would be complex and costly. One of the primary goals of this project is to design a cache that is simple and effective at improving the frame rate of the Pixel Router employing bilinear interpolation.

Section 3: Bilinear Interpolation and Memory Performance

Previously, the Pixel Router calculated the input pixel value to be mapped to the output pixel through a process called nearest neighbor interpolation. When computing the mapping between input and output pixels after warping an image, floating point numbers with a non-zero value after the radix for the input pixel locations are produced. However, pixels by their nature are quantized color values in integer grids. In nearest neighbor interpolation, the floating point x and y locations in the pixel grid are rounded to the nearest whole number to produce exact pixel locations. When implemented using pre-defined look-up tables, as in the Pixel Router, this process eliminates the need for any calculation to be performed by the graphics hardware and minimizes the required memory bandwidth. However, this method of interpolation is the least visually appealing, especially for content such as text and lines with rotation relative to the pixel grid. Below is an image used by Vijai Raghunathan that has been rotated and interpolated using the nearest neighbor method [3]. The text in this image, while readable, appears jagged and rough and below users' expectations for current graphics hardware.

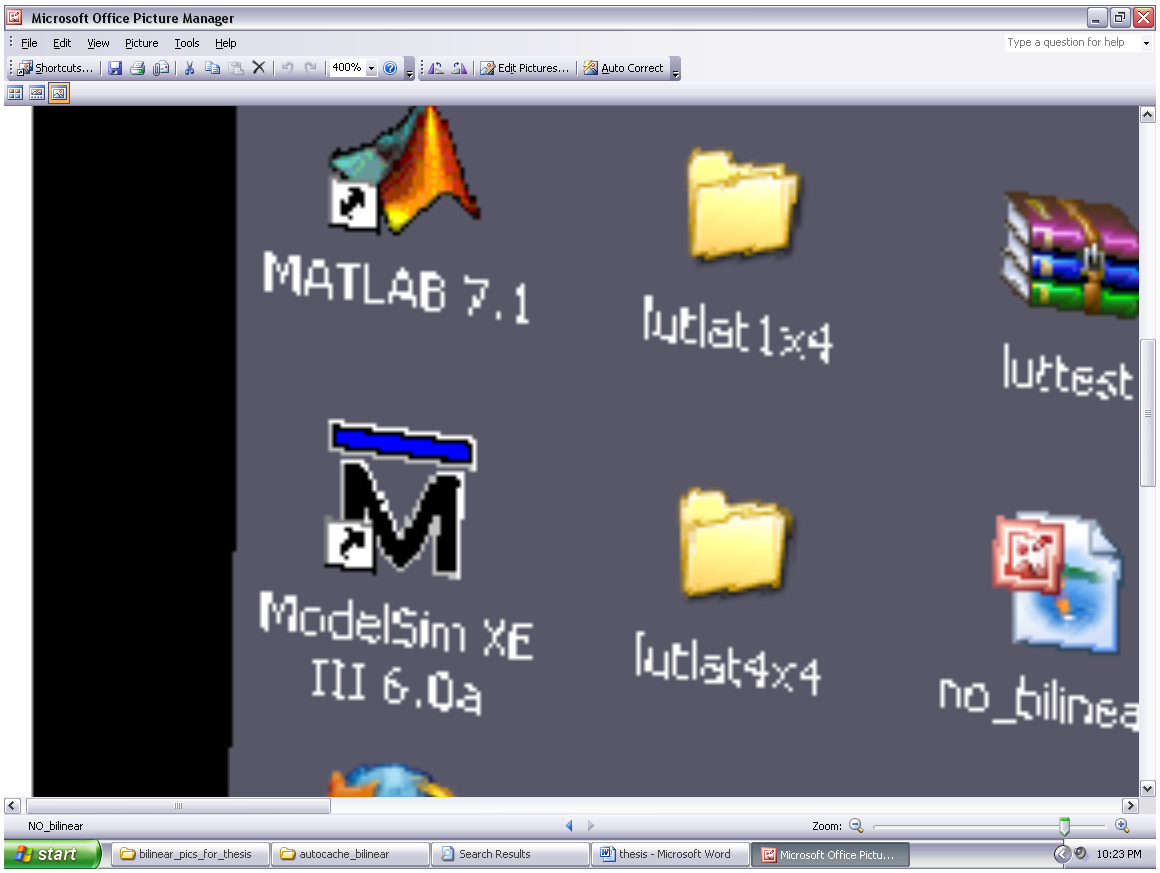

Figure 5 - Rotation with Nearest Neighbor Interpolation [3]

Bilinear interpolation is an image processing technique for improving the appearance of scaled, rotated, or warped images. Instead of rounding the floating point x and y coordinates of pixel values, the bilinear interpolation method computes a pixel value based on the weighted values of each of the 4 neighboring pixels [5]. This concept is illustrated in the image below.

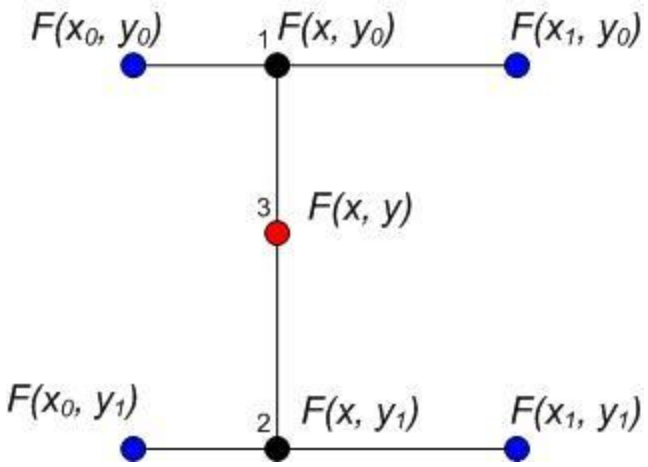

Figure 6 - Bilinear Interpolation Diagram

In the above image, the blue dots represent input pixels, where X_0, X_1, Y_0, and Y_1 are integer values representing pixel locations in the image. The function $F(X_a, Y_b)$ represents the color values of these pixels. The location (X,Y) is the interpolated input pixel location that maps to an output pixel location after an image warping operation, where X and Y are floating point, non-integer values. The two black dots represent values that are linearly interpolated along the X-direction. The equations below describe the calculations to compute the value of F(X, Y) by first linearly interpolating in the X-direction, and using the interpolated values to linearly interpolate in the Y-direction. In the context of an image, the distance between X_0 and X_1 is one pixel, and the same is true in the Y direction. That assumption is made in these equations.

$$\boldsymbol{f(x, y_0) = (x_1 - x) * f(x_0, y_0) + (x - x_0) * f(x_1, y_0)}$$

$$\boldsymbol{f(x, y_1) = (x_1 - x) * f(x_0, y_1) + (x - x_0) * f(x_1, y_1)}$$

$$\boldsymbol{f(x, y) = (y_1 - y) * f(x, y_0) + (y - y_0) * f(x, y_1)}$$

This three step equation can be simplified in to one by substituting the equations for $F(X, Y_0)$ and $F(X, Y_1)$ into the final equation.

In order to perform the bilinear interpolation calculation, the FPGA must be provided with a base input pixel address (u, v) and a value for both the x and y directions that represent that percentage of pixels (u+1, v), (u, v+1), and (u+1, v+1) to use. It was determined by Vijai Raghunathan that providing 3 bits after the radix point for both the x and y floating point values yielded sufficient resolution to produce visually appealing results. The bits after the radix point correspond to the values of (x1 – x), (x – x0), (y1 – y), and (y – y0), and determine the precision of the bilinear interpolation calculation. Figure 7 below is an image produced by Vijai Raghunathan rotated with bilinear

interpolation calculated using only 3 binary digits after the radix [3]. Vijai found that increasing the number of bits after the radix beyond 3 provided limited benefit to the resulting image quality. Thus, to conserve bit space in the Look Up Table, 3 bits will be used for bilinear interpolation.

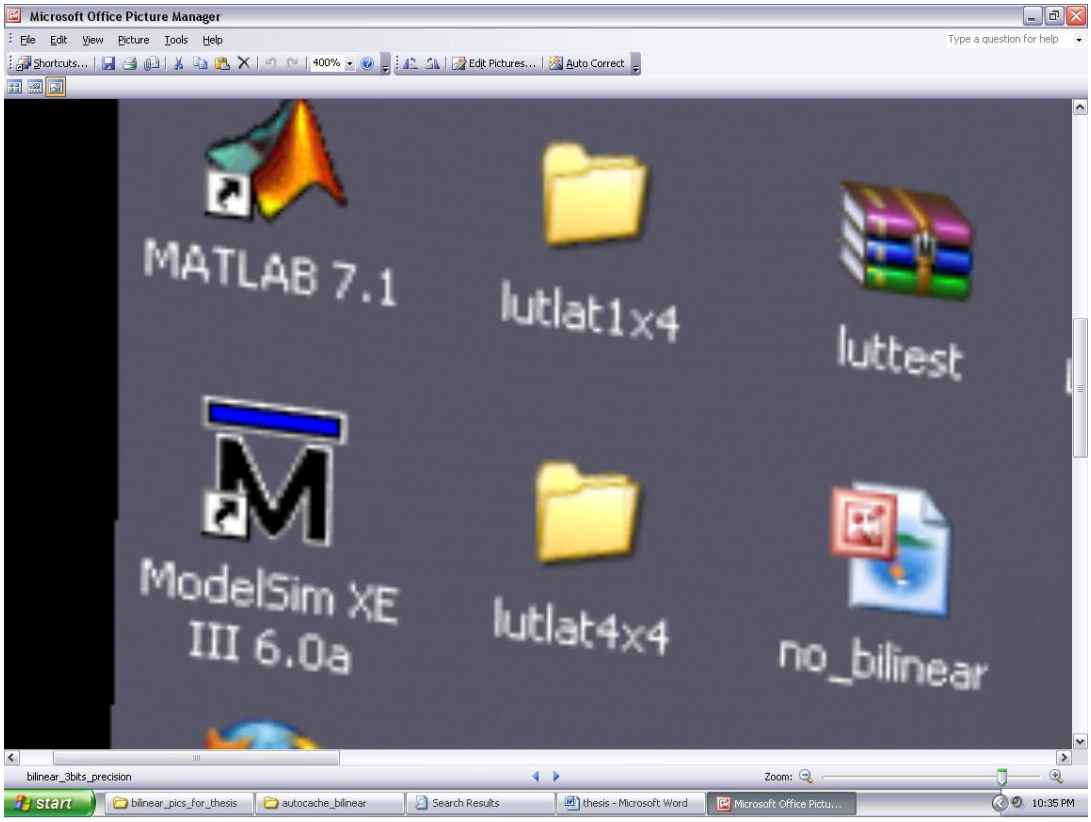

Figure 7 - Rotated Image with Bilinear Interpolation [3]

Memory Performance

Though bilinear interpolation provides great benefit to the appearance of rotated, warped, and scaled images, it is not without its cost. The first penalty is the calculation time required to compute the resulting output pixel from the 4 input pixels. The bilinear interpolation formula requires 4 multiplications and 4 additions to calculate the value of each output pixel. However, this performance penalty can be treated as negligible because this process can be pipelined through hardware design. The major performance penalty comes from having to read pixel values from 4 locations in Random Access Memory in order to compute the value of 1 output pixel. This effectively multiplies the required memory bandwidth by 4. Additionally, when accessing Random Access Memory, there is a latency penalty for opening a new memory row. Bilinearly interpolated pixels always require pixel data from 2 different rows, exposing this latency penalty on every pixel calculation. The Pixel Router is using

Micron MT4632M16P-5B devices for the memory banks. These devices are set to operate at 133MHz, and are programmed to use 2.5 cycle CAS latency with a burst length of 2 [3]. Based on the datasheet, the row opening penalty is due to latency values, the Auto Refresh command (^{t}RFC) and the Precharge command (^{t}RP). ^{t}RFC is specified at approximately 10 clock cycles and ^{t}RP at 2 clock cycles for a penalty of 12 clock cycles each time a new row is accessed [6]. The image below, which is taken from the data sheet, shows the low-level signals required to access a memory location and the latency associated with such access.

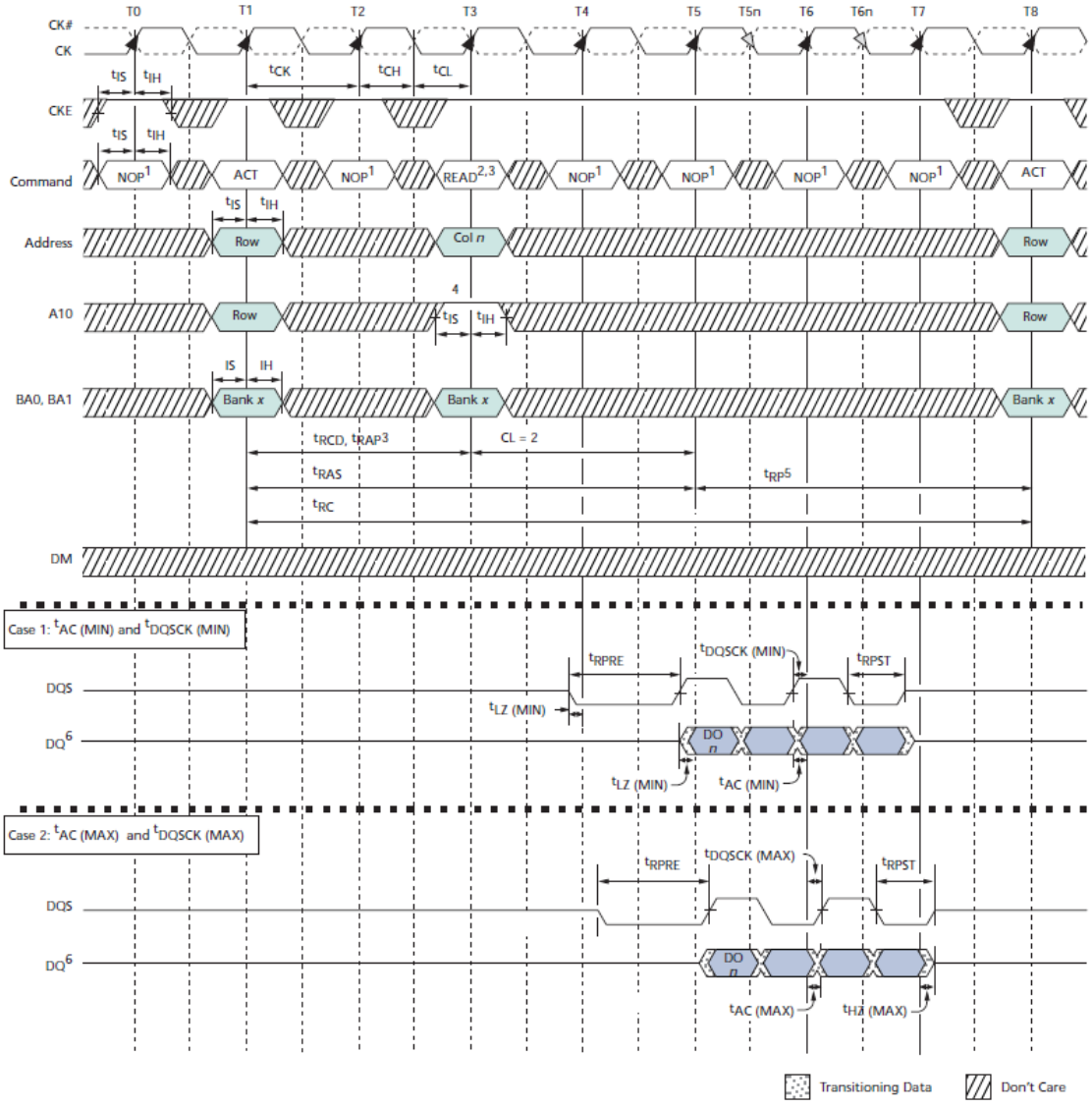

Figure 8 - SDRAM Read Timing Diagram [6]

If no action is taken to mitigate this latency when bilinear interpolation is implemented on the Pixel Router, the result is a dramatic decrease in performance (measured in output frame rate). The current hardware operates at 20 frames per second using nearest neighbor interpolation. The simulation by Vijai Raghunathan in the previous chapter suggests that the implementation of bilinear interpolation with no memory caching will cause the access time for one 1024x768 frame to drop to approximately 100ms. This equates to a frame rate on four 1024x768 output channels of 2.5 frames per second. Such a low frame rate would render the Pixel Router useless for virtually all applications besides the display of still images. The goal of the Pixel Router project is to produce a commercially viable product, and thus there cannot be a tradeoff between acceptable image quality and acceptable frame rate; both goals must be met. Therefore a cache system must be designed to allow the Pixel Router to process image transformations using bilinear interpolation at a frame rate that meets users' expectations.

Section 4: Prefetch Cache Design and Simulation

Basic Cache System

A common method of mitigating performance drops due to memory latency is to introduce a cache in the system. A cache serves as a small block of memory that contains a subset of the data stored in a larger memory block, and can be accessed much more quickly. Because the cache only contains a small amount of available memory, it should be loaded with data that is likely to be used on the next read attempt. Of course, the memory address location that is requested cannot always be guaranteed to be available in the cache because it is likely that all or most of the memory address locations in the larger memory block will be accessed at some point. Thus, a cache design is often evaluated by its efficiency or hit rate, which is defined as the percentage of memory access requests where the data is available in the cache versus the total number of memory access requests.

The design of the Pixel Router lends itself well to a cache system because the data in the input memory is always accessed in the same order. The access pattern is described by Look Up Tables (LUTs) which are calculated offline and loaded into the Pixel Router's memory at runtime. Because there is a much larger clock cycle penalty for accessing new memory rows than for new memory columns, the goal of a cache design for the Pixel Router should be to minimize the number of rows that need to be opened in order to read one video frame from memory. The previous work by Vijai Raghunathan described earlier focused on improving the performance of the pixel router under the worst-case scenario, which is a 45 degree rotation image transformation. Under that scenario, each memory access attempt would request pixel data from a new row and column in memory. His design proved effective at enhancing the performance of the Pixel Router under that condition, however the memory block system would have been costly and difficult to implement. Additionally, a 45 degree image rotation is not a typical scenario for a multi projector display. While projectors that are to be blended via software calibration can be placed arbitrarily, often they are at least "casually aligned" and placed in a generally normal, horizontal configuration. Often, the average rotation amount for each projector is quite small. The cache design for the Pixel Router will remove the constraint of rotation independence and instead focus on optimization for cases of image rotation of 5 degrees or less. Thus, a simpler cache system can be designed that is highly efficient for small amounts of rotation, which covers a majority of applications. Simulations will be conducted with up to 45 degrees of rotation, however, to fully evaluate any cache design.

The primary goal of the cache for the Pixel Router is to improve the frame rate for performing bilinear interpolation. Because bilinear interpolation requires two rows of input pixel data, the cache should contain at least two rows of data to avoid row opening penalties on every pixel read. Below are tables showing simulations of the Pixel Router using bilinear interpolation with simple two row caches of various widths with different amounts of rotation on the input image. Hennessy and Patterson [7] describe

several areas in which to optimize a cache. These include reducing the hit time, increasing the cache bandwidth, reducing the penalty for a cache miss, reducing the cache miss rate, and adding parallelism to the cache architecture. In these simulations, the effectiveness of the cache is evaluated by three metrics: frame rate, hit rate, and input memory efficiency. The ultimate goal is to increase the frame rate on the device, and the effectivenss of the cache is determined by the hit rate and input memory efficiency. The Frame Rate calculation uses the Pixel Router's clock speed of 133MHz. The assumptions in this calculation are that the cache cannot be written to and read from at the same time, and that it takes one clock cycle to read one interpolated pixel from the cache (4 data locations due to bilinear interpolation). The Pseudocode below describes the process for determining the total number of clock cycles to render one frame of data.

```
for each (CacheLine_Refill) {
     if(currentRow = lastRowOpened)
          loadCycles = CACHE_WIDTH * T_READ_PIXEL
     else
          loadCycles   =   T_OPEN_ROW   +   CACHE_WIDTH   *
          T_READ_PIXEL

     Total_Cycles   =   Total_Cycles   +   loadCycles   +
     pixelsProcessed
}
```

This code reflects the behavior of the SDRAM by including the clock cycle penalty for opening a new row of data. This calculation also assumes that the cache cannot be written to and read from at the same time by adding the `pixelsProcessed` value to the number of cycles calculated. The number of clock cycles calculated by this (`Total_Cycles`) refers to the cycles required for one frame of data on one video output. The Pixel Router uses 4 video outputs that are written to in round-robin fashion, so to estimate the real frame rate this number must be multiplied by 4 (assuming all video outputs have the same image transformation/rotation). The clock cycle value listed in the tables below is calculated before this multiplication. The frame rate is calculated as

$$Frame\ Rate = \frac{1}{4 * Clock\ Cycles} * Clock\ Speed$$

Input memory efficiency is related to the frame rate and describes how effectively the SDRAM for the input frame buffer is being utilized. It can be calculated using the ratio of pixels read versus the total number of pixels possible to be read, and an efficiency of 100% would mean that a pixel is read from the SDRAM at every available instance. For DDR SDRAM operating at 133MHz, that corresponds to 266 million pixels read per second. Row opening wait times, during which no input pixels can be read, reduces the memory efficiency. The memory efficiency can be calculated based on the output frame

rate for cases of regular image transformation. As discussed previously, when applying bilinear interpolation to the image 4 input pixels are required to calculate each output pixel. However, for a regular image rotation there are 2 unique pixels used per output pixel, as there is overlap between the input pixels required to calculate each output pixel. Therefore, input memory efficiency on DDR SDRAM for the Pixel Router can be calculated as

$$Input\ Memory\ Efficiency = \frac{Pixels\ per\ Frame * 4 * 2 * Frame\ Rate * 100\%}{Clock\ Speed * 2}$$

The cache hit rate describes how effectively the data within the cache is utilized by showing the number of times the cache is accessed where the desired data is in the cache versus the number of times where that data is not in the cache. It is calculated as

$$Hit\ Rate = \frac{Cache\ Hits * 100\%}{Cache\ Hits + Cache\ Misses}$$

The tables below show the results of the simulations of the initial cache design, evaluated using the metrics described above.

Table 1 - 16 Pixel Cache Simulation

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	3171612	10.48	92.77	24.80
1	3358048	9.90	92.05	23.42
2	3172687	10.48	92.14	24.79
3	3953832	8.41	90.05	19.89
4	2793343	11.90	92.39	28.15
5	2859129	11.63	91.95	27.51
10	5166938	6.44	85.46	15.22
15	6999775	4.75	80.34	11.24
20	8390303	3.96	76.48	9.37
25	9368092	3.55	73.80	8.39
30	9960016	3.34	72.23	7.90
35	10190250	3.26	71.71	7.72
40	10080609	3.30	72.16	7.80
45	9651829	3.44	73.53	8.15

Table 2 - 32 Pixel Cache Simulation

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	2355396	14.12	96.58	33.39
1	2268204	14.66	96.52	34.67
2	2091471	15.90	96.56	37.60
3	3067752	10.84	94.95	25.64
4	3993127	8.33	93.42	19.69
5	4883897	6.81	91.95	16.10
10	8824890	3.77	85.46	8.91
15	11948447	2.78	80.34	6.58
20	14310431	2.32	76.48	5.50
25	15961532	2.08	73.80	4.93
30	16947728	1.96	72.23	4.64
35	17310474	1.92	71.71	4.54
40	17087169	1.95	72.16	4.60
45	16313397	2.04	73.53	4.82

Table 3 - 64 Pixel Cache Simulation

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	1974900	16.84	98.34	39.82
1	1944180	17.10	98.25	40.45
2	3822415	8.70	96.56	20.57
3	5609832	5.93	94.95	14.02
4	7303463	4.55	93.42	10.77
5	8933433	3.72	91.95	8.80
10	16140794	2.06	85.46	4.87
15	21845791	1.52	80.34	3.60
20	26150687	1.27	76.48	3.01
25	29148412	1.14	73.80	2.70
30	30923152	1.08	72.23	2.54
35	31550922	1.05	71.71	2.49
40	31100289	1.07	72.16	2.53
45	29636533	1.12	73.53	2.65

Table 4 - 128 Pixel Cache Simulation

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	1483583	22.41	99.12	53.01
1	2822772	11.78	98.25	27.86
2	5553359	5.99	96.56	14.16
3	8151912	4.08	94.95	9.65
4	10613799	3.13	93.42	7.41
5	12982969	2.56	91.95	6.06
10	23456698	1.42	85.46	3.35
15	31743135	1.05	80.34	2.48
20	37990943	0.88	76.48	2.07
25	42335292	0.79	73.80	1.86
30	44898576	0.74	72.23	1.75
35	45791370	0.73	71.71	1.72
40	45113409	0.74	72.16	1.74
45	42959669	0.77	73.53	1.83

Several conclusions can be drawn from this data. The first is that, in terms of frame rate, none of these is an acceptable cache design. While the cache is providing some benefit, the frame rate for all but one case is less than 20 fps. This is despite the fact that the hit rate for each cache is very high. All 4 cache widths have hit rates of over 90% for image rotations of between 0 and 5 degrees. Another conclusion is that generally smaller cache widths perform better for higher amounts of image rotation and larger cache widths perform better for lower amounts of image rotation. A graph showing the frame rates of all 4 cache design is shown below.

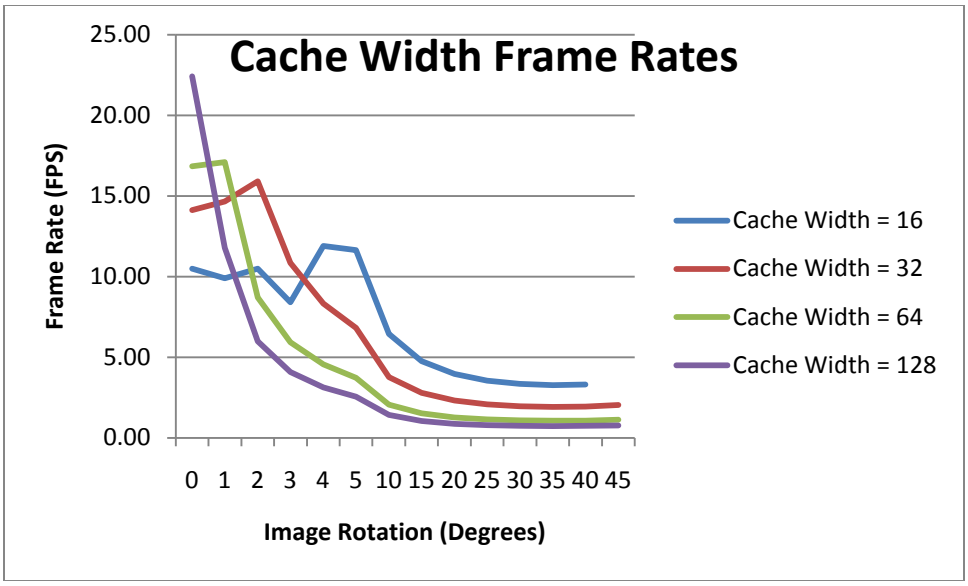

Figure 9 - Cache Simulation Comparing Cache Width

Another interesting aspect of this cache design that can be drawn from the data is that, for smaller cache widths especially, the frame rate is lower for no rotation than it is for small amounts of rotation (1-4 degrees generally). Two factors contribute to this effect. The first is that there is an advantage when the row desired to be loaded in to the cache is the same as the previous row opened, and thus the row opening penalty does not apply as the row is already precharged. Two separate rows are used on each cache load, but for small amounts of positive (counter-clockwise) rotation, as was simulated, the first line loaded in the current cache reload is the same row as the second line loaded in the previous cache load. For instance, the cache might contain data from rows 0 and 1, and when it is refreshed contain data from rows 1 and 2. In these simulations, images with no rotation did not enjoy this advantage because rows would always be loaded in alternating order (e.g. 0, 1, 0, 1, 0, 1, etc.). With some effort, the same data rows could be loaded in the cache in the order 0, 1, 1, 0, 0, 1, etc., maximizing the efficiency of the memory in this case. Because this was not simulated in this matter, 0 degree image rotation had in some cases significantly lower frame rate than small amounts of image rotation.

As an aside, something similar to a "resonant frequency" of the cache can be determined by describing image rotation (a continuous function) using the number of pixels (discrete values) in the horizontal direction before a change in row value occurs. For a given image rotation amount, θ, this can be calculated using

$$\tan\theta = \frac{\Delta r}{\Delta c}$$

where Δr is the change in pixel row value, and Δc is the change in pixel column value. For angles less than 45 degrees, Δr can be assumed to be 1. The caches described earlier will operate most efficiently at the amount of image rotation with a

corresponding Δc close to but below their cache width. This means that few pixels at the end of the cache will go unused because they belong to the wrong row, resulting in fewer wasted clock cycles loading unnecessary pixels. Below in Table 5 is a list of the frequencies that were simulated, along with their corresponding Δc. A rotation of 0 degrees would of course have an infinite Δc.

Table 5 - Optimized Cache Sizes for Rotation

Rotation (degrees)	Δc
1	57
2	29
3	19
4	14
5	11
10	6
15	4
20	3
25	2
30	2
35	1
40	1
45	1

This means that a cache of width 64 will operate most efficiently for images rotated approximately 1 degree and a cache of width 32 will operate most efficiently for images rotated approximately 2 degrees. A cache width of 16 presents an interesting scenario for images rotated 3 degrees. Because Δc is 19, if 16 usable pixels are loaded in to the cache in one cache refill, only 3 usable pixels will be loaded in to the cache on the next cache refill because the image row changes on the 20^{th} pixel. Thirteen pixels loaded into the cache1 go unused in this cache refill. Only 59% of the pixels loaded in the two cache refills are useful, and this perhaps more accurately describes this cache's effectiveness than hit rate percentage.

Prefetch Cache System

Clearly, there exists room for improvement in the design of this cache system. A major area of inefficiency lies in the fact that the cache cannot be written to and read from at the same time. This can be addressed by exploring the concept of "prefetching". The pre-calculated LUT architecture of the Pixel Router allows for knowledge of the exact input pixel locations where the cache will need to be refilled. This gives rise to the ability to create a "ping-pong" cache system, where one cache buffer is being read from while another cache buffer is being loaded with the next set of pixels to be used. The net effect of this is a two stage cache pipeline system that eliminates the inefficiency of

being unable to read from and write to the cache at the same time. The image below illustrates this concept with an example cache width of 8 pixels.

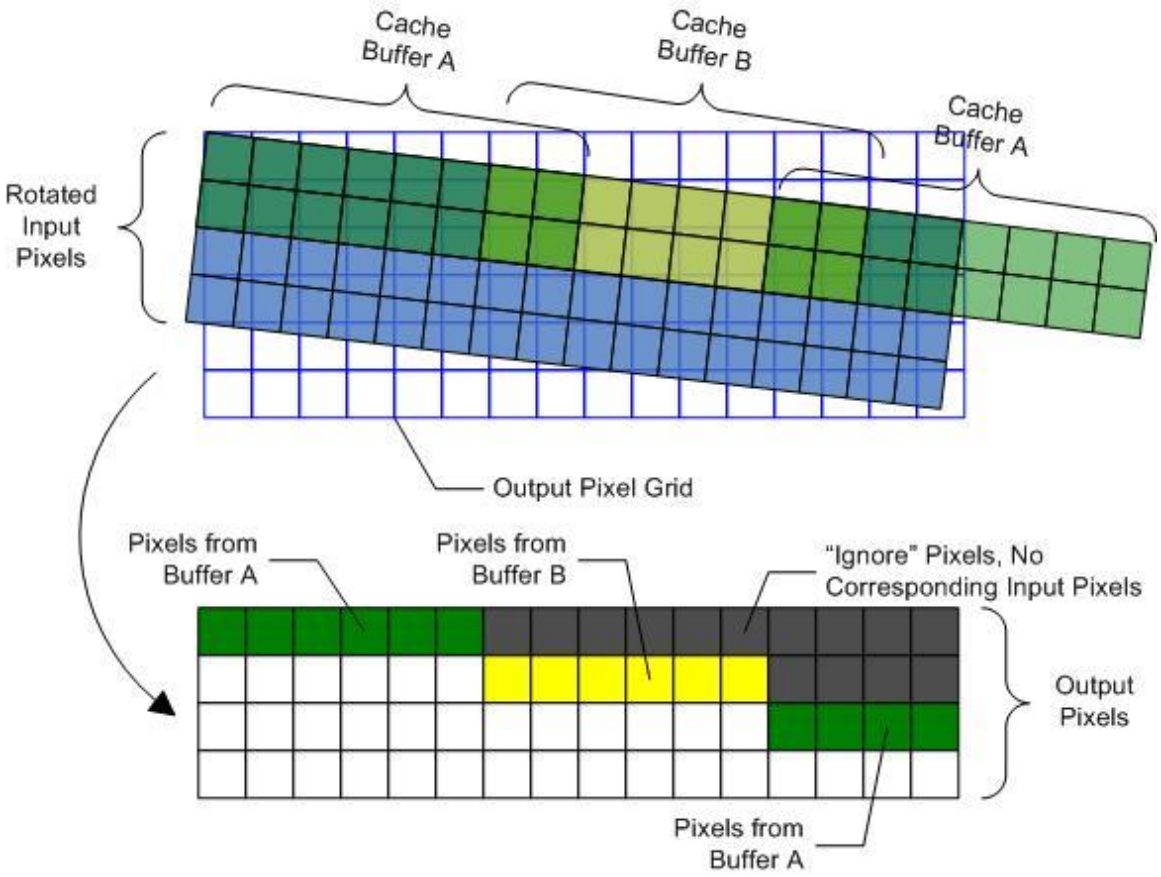

Figure 10 - Cache Refill Example

In the above image, the blue outlined squares represent the output pixel space, and the rotated blue shaded squares represent input pixels in memory that get rotated in the image transformation. The input image in this case is 16x4 pixels, and each cache buffer contains 8x2 pixels. The image represents a simulation of 3 separate cache fills, first to Buffer A, then to Buffer B, then back to Buffer A. The lower half of the image shows the input pixels applied to the output image space after bilinear interpolation. The green pixels in the lower half of the image were interpolated from the input pixels in Buffer A, and the yellow pixels were interpolated from the input pixels in Buffer B. The dark grey pixels are output pixels which have no corresponding input pixels due to the rotation and thus are ignored for processing and show up as black pixels on the output.

This prefetching cache system was simulated in the same method as the non-prefetch cache system. Again, in these simulations only each cache buffer contains only 2 rows of pixels. The results of the simulation are shown in the tables and graph below.

Table 6 - 16 Pixel Cache Simulation with Prefetching

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	2262312	14.70	92.77	34.76
1	2357228	14.11	92.05	33.36
2	2183259	15.23	92.14	36.02
3	2701524	12.31	90.05	29.11
4	1835531	18.11	92.39	42.84
5	1846757	18.00	91.95	42.58
10	3337974	9.96	85.46	23.56
15	4525451	7.35	80.34	17.38
20	5430239	6.12	76.48	14.48
25	6071372	5.48	73.80	12.95
30	6466160	5.14	72.23	12.16
35	6630138	5.01	71.71	11.86
40	6577329	5.06	72.16	11.96
45	6321045	5.26	73.53	12.44

Table 7 - 32 Pixel Cache Simulation with Prefetching

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	1495236	22.24	96.58	52.60
1	1391436	23.90	96.52	56.52
2	1225999	27.12	96.56	64.15
3	1796712	18.51	94.95	43.77
4	2337959	14.22	93.42	33.64
5	2859129	11.63	91.95	27.51
10	5166938	6.44	85.46	15.22
15	6999775	4.75	80.34	11.24
20	8390303	3.96	76.48	9.37
25	9368092	3.55	73.80	8.39
30	9960016	3.34	72.23	7.90
35	10190250	3.26	71.71	7.72
40	10080609	3.30	72.16	7.80
45	9651829	3.44	73.53	8.15

Table 8 - 64 Pixel Cache Simulation with Prefetching

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	1139316	29.18	98.34	69.03
1	1065588	31.20	98.25	73.80
2	2091471	15.90	96.56	37.60
3	3067752	10.84	94.95	25.64
4	3993127	8.33	93.42	19.69
5	4883897	6.81	91.95	16.10
10	8824890	3.77	85.46	8.91
15	11948447	2.78	80.34	6.58
20	14310431	2.32	76.48	5.50
25	15961532	2.08	73.80	4.93
30	16947728	1.96	72.23	4.64
35	17310474	1.92	71.71	4.54
40	17087169	1.95	72.16	4.60
45	16313397	2.04	73.53	4.82

Table 9 - 128 Pixel Cache Simulation with Prefetching

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	1041215	31.93	99.12	75.53
1	1944180	17.10	98.25	40.45
2	3822415	8.70	96.56	20.57
3	5609832	5.93	94.95	14.02
4	7303463	4.55	93.42	10.77
5	8933433	3.72	91.95	8.80
10	16140794	2.06	85.46	4.87
15	21845791	1.52	80.34	3.60
20	26150687	1.27	76.48	3.01
25	29148412	1.14	73.80	2.70
30	30923152	1.08	72.23	2.54
35	31550922	1.05	71.71	2.49
40	31100289	1.07	72.16	2.53
45	29636533	1.12	73.53	2.65

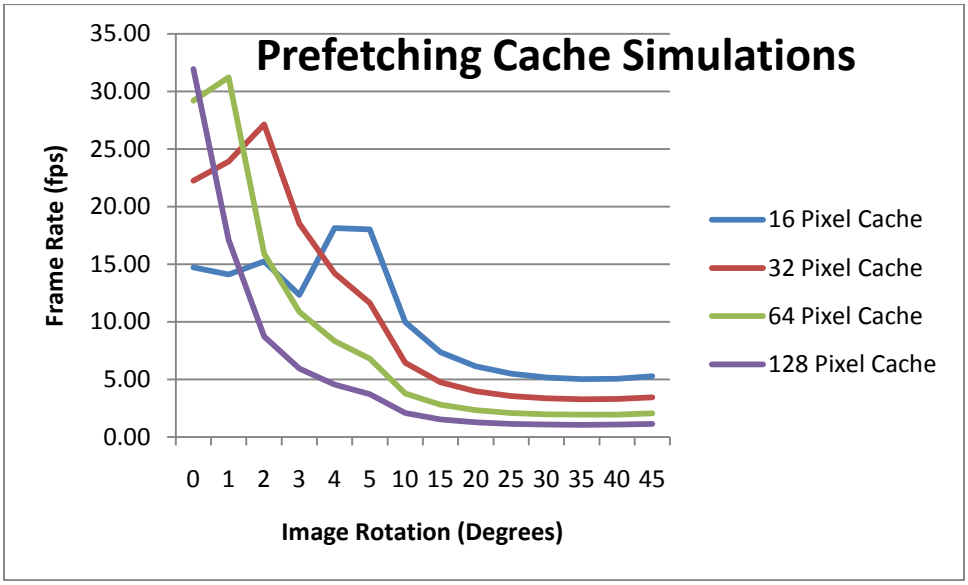

Figure 11 - Cache Simulation with Prefetching

Based on these simulations, the frame rate using a prefetching cache system is greatly improved over using a non-prefetching cache. One item to note, however, is that the cache hit rate is unchanged between the two cache designs. This is because cache hit rate is defined as

$$Cache\ Hit\ Rate = \frac{Cache\ Misses}{Cache\ Hits + Cache\ Misses} * 100\%.$$

The number of cache hits and cache misses is unaffected by adding a second cache buffer. The only operational difference is that when a cache miss occurs, the buffers are switched if the second buffer is done loading. The table below shows the percentage improvement of the prefetching cache over the non-prefetching cache for the four cache widths examined.

Table 10 - Prefetching Cache Improvement

Cache Width (Pixels)	Average Improvement (%)
16	50.95
32	68.89
64	81.91
128	45.03

Dynamically Loaded Cache System

This cache design brings the Pixel Router operation with bilinear interpolation closer in line to the previous performance standard using nearest neighbor interpolation of 20 frames per second. In fact, several cache widths exceed this mark for small amounts of rotation. However, it is clear that the performance of this cache is very dependent on the image transformation to be performed. While the goal of this project was to design a cache that is optimized for small amounts of rotation, a robust design should make every effort to maximize performance in as many cases as possible while still maintaining its optimization target. The prefetching cache design reduced or eliminated the clock cycle penalty of reading data from the cache. However as the amount of image rotation is increased, more of the pixels that are loaded in to the cache remain unused. Once again, the fixed nature of the LUT architecture of the Pixel Router lends itself to a solution to this issue. Just as it is possible to know which memory address in the input buffer to begin loading the second cache buffer with, it is also possible to know which pixels in the cache will cause a cache miss. Thus, an additional field can be added to the look up table that tells the Pixel Router how many pixels to load in the cache on a given cache refill. This "Pixels to Load" field allows the cache to be dynamically loaded with only pixels that are guaranteed to be used rather than with an entire set of pixels equal to the cache width.

To determine the impact of a dynamically loaded cache system, a 64 pixel wide cache was examined. The table below shows the simulation results of the cache simulated under the same conditions as the previous cache designs. The following graph compares the frame rates of the dynamically loaded cache with the 64 pixel statically loaded cache with and without prefetching.

Table 11 - Dynamically Loaded Cache Simulation

Rotation (Degrees)	Clock Cycles	Frame Rate (fps)	Hit Rate (%)	Memory Eff. (%)
0	1114740	29.83	98.34	70.55
1	986036	33.72	98.25	79.76
2	1171737	28.38	96.56	67.12
3	1350784	24.62	94.95	58.22
4	1518483	21.90	93.42	51.79
5	1679797	19.79	91.95	46.82
10	2652096	12.54	85.46	29.65
15	3597563	9.24	80.34	21.86
20	4320215	7.70	76.48	18.20
25	4835102	6.88	73.80	16.27
30	5155964	6.45	72.23	15.25
35	5295096	6.28	71.71	14.85
40	5263599	6.32	72.16	14.94
45	5072001	6.56	73.53	15.51

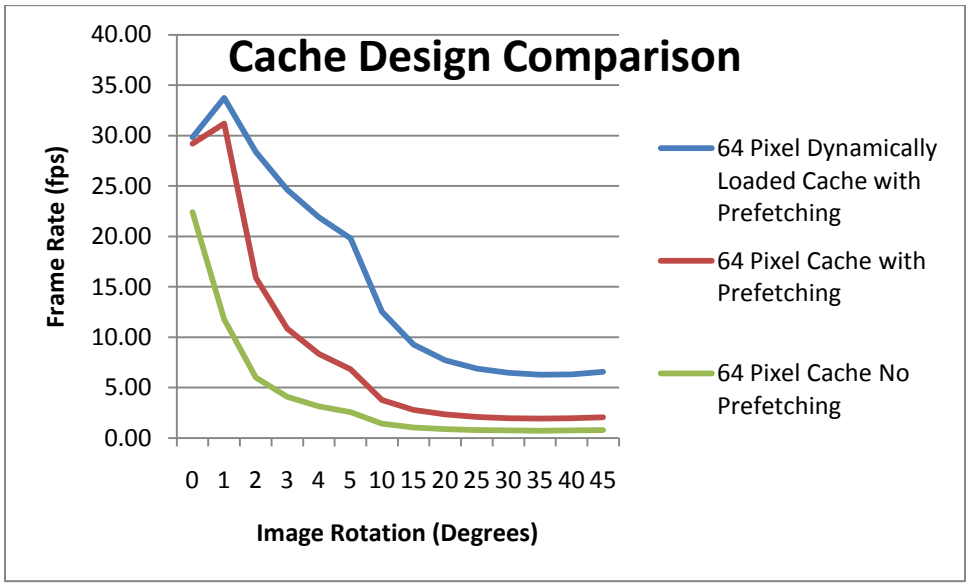

Figure 12 - Dynamically Loaded Cache Simulation

Dynamically loading this cache with only pixels that are guaranteed to be useful provides an enormous performance increase, especially for cases with image rotation. This makes sense as image rotation causes fewer pixels in a given input row to be valid for a given output row. Dynamically loading the cache brings any given cache width in line with the values in Table 5, where the effective cache width is roughly equal to Δc for a given image rotation value. Based on these simulations, the Pixel Router should

operate at or above 20 frames per second for up to 5 degrees of rotation for a 1024x768 pixel image.

Cache Definitions

The cache design implemented in the Pixel Router to support bilinear interpolation was a 64 pixel wide dynamically loaded cache with prefetching. The image below presents this cache and the definitions of its fields.

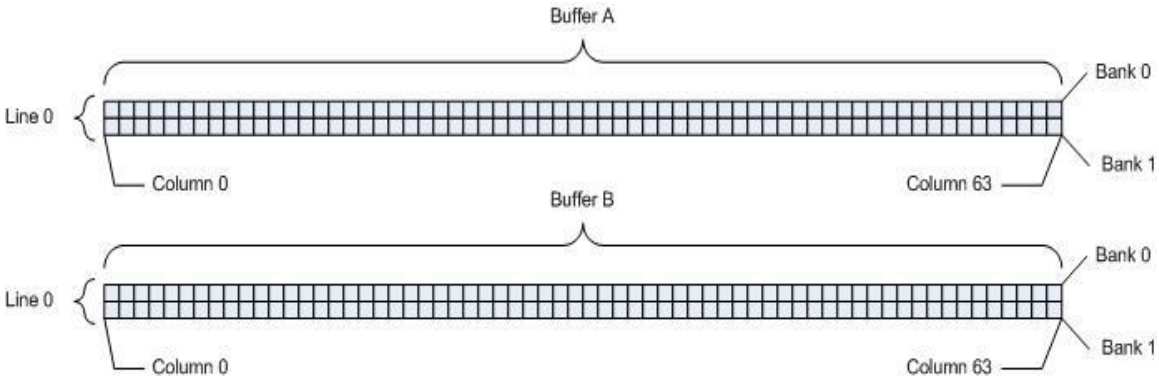

Figure 13 – Cache Field Definitions Definitions

The cache is comprised of two buffers, Buffer A and Buffer B, which act as the prefetch pipeline. One buffer is being read from while another buffer is being written to. Each cache buffer consists of one cache line (Line 0). A cache line contains data from two rows of input pixels, however this data is only valid for one output pixel line due to bilinear interpolation, thus one line represents two rows. The input pixel rows in each cache line are referred to as banks, and each cache line contains two of them (Bank 0 and Bank 1). There are 64 columns in each bank which represent the 64 pixel locations available.

Additional Cache Lines

Cache simulations to this point have included one cache line per buffer. The natural extension of this is to consider whether additional cache lines would provide performance benefits. To determine this, an 8 line cache was examined. In this case, Buffer A and Buffer B would each contain 8 cache lines, and each cache line would be filled starting with the address of the next predicted cache miss. The simulation showing the percentage improvement in frame rate of an 8 line cache over a 1 line cache are shown below.

Table 12 - Cache with 8 Lines

Rotation (Deg)	Frame Rate, 8 Lines (fps)	Frame Rate, 1 Line (fps)	Improvement (%)
0	38.81	29.83	30.13
1	33.50	33.72	-0.66
2	28.69	28.38	1.09
3	25.08	24.62	1.90
4	22.48	21.90	2.64
5	20.43	19.79	3.21
10	13.11	12.54	4.55
15	9.72	9.24	5.13
20	8.13	7.70	5.66
25	7.31	6.88	6.27
30	6.90	6.45	6.98
35	6.77	6.28	7.87
40	6.88	6.32	8.98
45	7.24	6.56	10.47

While the 8 line cache offers improvement for the case of 0 degree rotation, there is little difference in performance for images with rotation. The improvement in the 0 degree rotation case can be attributed to the fact that this cache is organized and written in such a way as to minimize row opening penalties. To achieve this, all entries of a particular row within a cache refill are opened consecutively, regardless of the order they will be accessed in. This has the greatest impact on images with no rotation because most of the entries in a particular cache refill are in the same video row, thus eliminating a greater number of row opening penalties. For rotated images, however, roughly the same number of row opening penalties occur in an 8 line cache as in a 1 line cache, which is why there is little if any improvement gained by using an 8 line cache in this case. Because the overall benefit of additional cache lines is minimal, it was decided that a 1 line cache would suffice in the final implementation of the Pixel Router to reduce hardware complexity and conserve memory space on the FPGA.

Cache Operation Example

To better understand the process of writing to and reading from a cache, here a clock cycle simulation is presented as an example. This simulation uses data from a look up table that describes a 3 degree image rotation. Below is the data from that look up table.

Table 13 - Prefetch LUT Data

Pixels to Load	Input Column	Input Row	Cache Line	Bank
20	0	0	0	0
20	0	1	0	1
22	18	1	0	0
22	18	2	0	1
20	38	2	0	0
20	38	3	0	1
22	56	3	0	0
22	56	4	0	1

There are a few points to note about this data. The Pixels to Load field in this data, as in all cases, is an even value. This is due to the fact that the DDR SDRAM data bus to the FPGA is effectively 64 bits per clock cycle, which is enough data for two 24-bit pixels. Thus, it does not make sense to load an odd number of pixels because two pixels are loaded per clock cycle no matter what. In addition, the Input Column field is also even. This will be addressed in a later section, but to save space in the Look Up Table, the input address field was reduced by one bit, forcing the column of the address to be an even number. The image below shows the timing diagram of the memory reads and writes for this simulation.

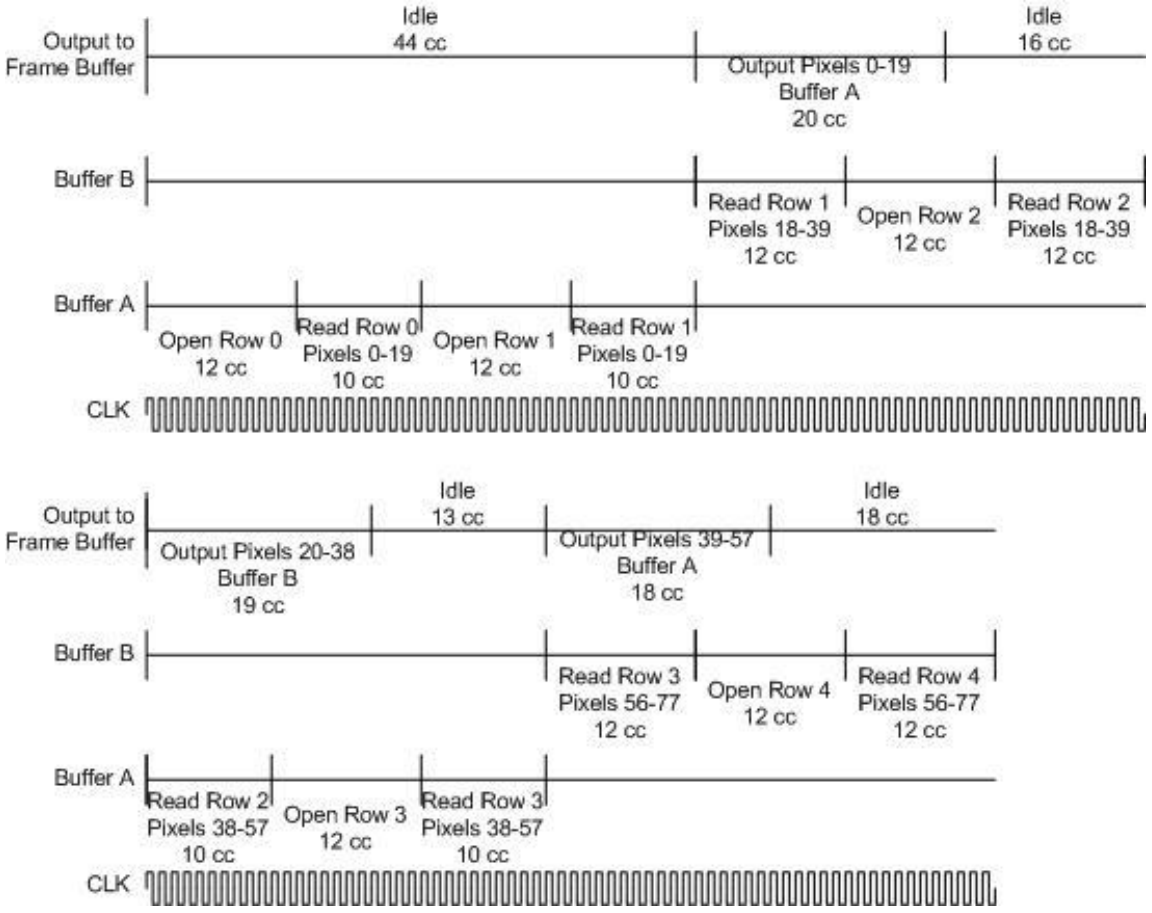

Figure 14 - Cache Operation Example Timing Diagram

This conceptual timing diagram illustrates the impact of the memory overhead and the effect of the "ping-pong" cache. The first line represents pixels output to the frame buffer, where they are stored before being output on the HDMI port. The next two lines tell what activity is taking place on either Buffer A or Buffer B, and how many clock cycles it takes. The last line is the Clock signal for reference. Note that after the first time Buffer A is written to, the amount of time the output to the frame buffer is idle drops significantly. Starting on the 45th clock cycle, the output frame buffer is written to for 57 clock cycles and idle for 47 clock cycles, giving it an active percentage of 54.8%. The first 44 clock cycles can be considered a one-time penalty at the beginning of the frame to load the first pixels into the first cache buffer, as after that output pixels will always be processed concurrently with cache refills.

Measured Results

This cache design has been implemented on the Pixel Router by Verien Design, Inc., an outside consulting firm, and is operational. The frame rate for LUTs of different rotation

amounts was measured to validate the results from simulation. The results are shown in the table below, along with the simulated values.

Table 14 - Measured versus Simulated Frame Rates

Rotation (Deg)	Real Frame Rate (FPS)	Simulated Frame Rate (FPS)
0	20	29.83
1	30	33.72
2	20	28.38
3	20	24.62
4	20	21.90
5	15	19.79
10	12	12.54

This frame rate data was taken from the Pixel Router control interface, which provides frame rate information for each input channel. The frame rate on Channel 1 was measured. The Pixel Router calculates frame rate in hardware and can only provide values that are divisors of 60, without rounding up. This explains the difference between the simulated and measured values. For instance, at 0 degrees rotation, the simulated value was 29.83 degrees, but the measured value in hardware rounded down to 20 degrees. Based on this measurement restriction, however, the measured frame rate correlates directly with the simulated frame rate. Unfortunately, measurements for images rotated more than 10 degrees were not able to be obtained due to a flaw in the hardware.

Section 5: LUT Description and Generation

LUT Description

The image transformation performed by the Pixel Router is calculated offline and described by Look Up Tables (LUTs) that tell the hardware what order to process the pixels in and the attenuation factor for specific pixels. To accommodate the prefetching cache system, a dual LUT architecture was developed. In this architecture, one LUT contains 2 entries for each cache refill (one for each row of input memory to be read) that describe which input pixels to load in the cache, how many to load, and which cache location they should be stored in. This LUT is known as the "Prefetch LUT". The other LUT, known as the "Per-Pixel LUT", has one entry per output pixel and contains information about where in the cache the input pixels are located, the attenuation factor of this output pixel, and the X and Y pixel weights for bilinear interpolation. Both the Prefetch LUT and Per-Pixel LUT are made up of 32-bit entries, with headers at the top that describe which type of LUT it is and how many entries are present. The Per-Pixel LUT contains as many entries as there are pixels in the output space. The number of entries in the Prefetch LUT is dependent on the number of cache refreshes required for a particular image transformation. The tables below are described in the Pixel Router Specification [3] and contain a detailed description of each LUT field.

Table 15 - Prefetch LUT Description [3]

Bits	Field	Description
31	Reserved	Reserved
30:26	Pixels_To_Load	Pixels to be loaded in to the cache, mod 2. The number of pixels loaded in to the cache is 2*(Pixels_To_Load+1).
25:24	Channel_ID	Input channel ID
23:4	Offset	Address of the first pixel to be loaded in to the cache line, mod 2. Address space allows for an image with 1920x1080 pixels
3	CD_Bank	Bank within Buffer A or B to which data will be written.
2:0	CD_ID	Cache line where the data will be written. Note that the Pixel Router currently only supports 1 cache line, so this value is always 0.

Table 16 - Per Pixel LUT Description [3]

Bits	Field	Description
31:30	Command	Command bits: 00: User bilinear interpolation, normal operation. 01: Switch cache buffers (A to B or B to A) due to cache miss.
29:28	Reserved	Reserved
27	CD_ID1	Cache Destination ID 1. Cache line in bank 1 containing pixel 3 and 4 for bilinear interpolation.
26	CD_ID0	Cache Destination ID 0. Cache line in bank 0 containing pixel 1 and 2 for bilinear interpolation.
25:20	ADDR1	Address of pixel 3 within cache line CD_ID1. Pixel 4 is located at ADDR1+1.
19:14	ADDR0	Address of pixel 1 within cache line CD_ID0. Pixel 2 is located at ADDR0+1.
13:6	Alpha	Alpha blending attenuation factor.
5:3	SubY	The Y coordinate of the output pixel relative to the input pixels for bilinear interpolation.
2:0	SubX	The X coordinate of the output pixel relative to the input pixels for bilinear interpolation.

The LUT entry fields were designed to maximize flexibility for any future upgrades by enabling control of multiple cache lines (only one is used currently) and leaving bits available for future use. For instance, the cache could be expanded to 128 pixels wide by using increasing the field size of ADDR1, ADDR0, and Pixels_To_Load by one bit using the bits that are currently labeled “Reserved” in the LUT description. Alternatively, additional cache lines could be added, causing the field size of CD_ID1 and CD_ID0 to increase. Through continued use of the Pixel Router and evaluation of future requirements, it will be determined which, if any, of these changes would provide maximum benefit or usefulness.

LUT Generation

Once the image transformation has been determined, either through calibration or a regular function such as image rotation, a two-pass approach is used to generate the Look Up Tables for the Pixel Router. The first pass transforms the output space of each output pixel to the input space via the image transformation function to correlate an input memory address for every output pixel. In the second pass, a “virtual cache” is created to simulate the cache operation in the Pixel Router. For each output pixel, it is determined whether or not the input pixel is present in the cache. If the pixel is not present, a new Prefetch LUT entry is created for that input memory address. A flowchart diagram describing the LUT generation process is shown below in Figure ***.

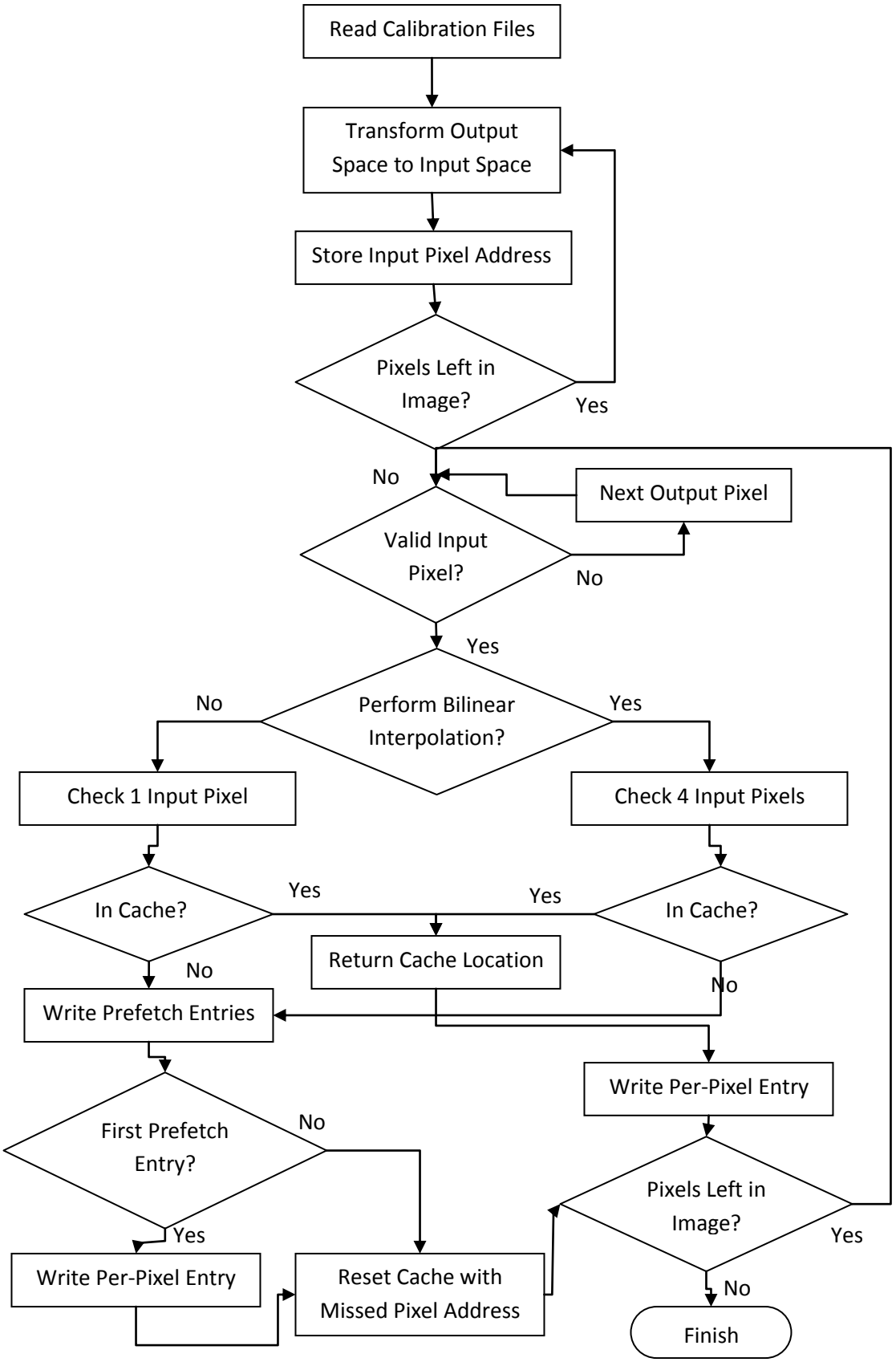

Figure 15 - LUT Generation Flowchart

Section 6: Packaging

Case and Power Supply

For successful marketing and commercialization, a product must be packaged in such a way that it is appealing as well as easy to use for end users. Mindful of this, and for the practical needs of easy and safe transportation, a 1U rack mount case was chosen and modified to house the Pixel Router. An off-the-shelf ATX power supply was placed in the case with the Pixel Router unit. An image of the interior of the rack mount case is shown below in Figure 16.

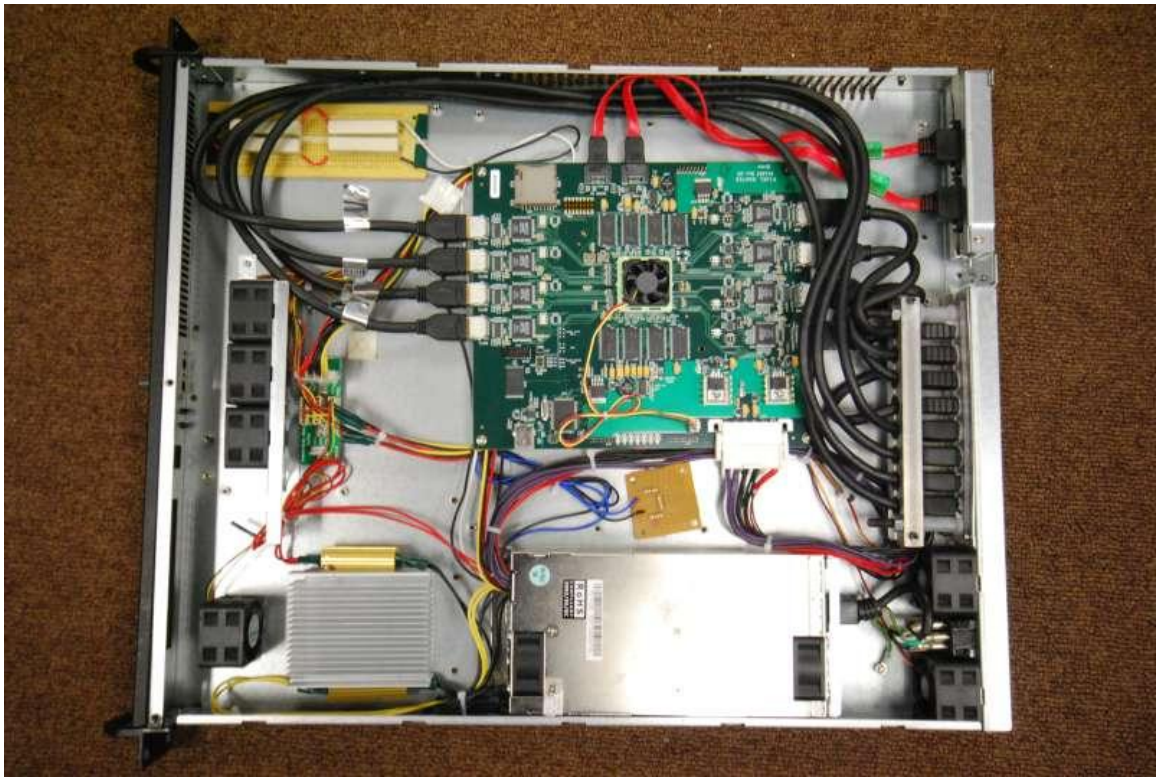

Figure 16 - Pixel Router Packaging

The case design presented a few physical and technical challenges. The first issue to address was making the HDMI ports available on the exterior of the case. Few devices use 8 HDMI ports in their design, and there was no off-the-shelf 8 port HDMI jack available for purchase. Therefore, in collaboration with the Center for Manufacturing Systems at UK, a custom device was developed to rigidly hold in place the female end of up to 8 male to female HDMI cables, with holes on the exterior side to allow the user to plug HDMI cables directly in to the box. The male ends of the HDMI cables connect to the HDMI jacks on the board itself, thereby extending the HDMI ports from the interior to the exterior of the box.

The 1U ATX power supply chosen to power the Pixel Router presented a technical challenge as well. ATX power supplies are designed to provide power to personal computer systems, which have much more varied voltage requirements than the Pixel Router. The Pixel Router is designed to be supplied with +5V and +3.3V for power. The ATX power supply provides output power at +12V, +5V, +3.3V, -5V, and -12V, and has minimum current requirements on each of these outputs. Below is a table with the voltage outputs and the respective minimum current requirements, per the data sheet for the power supply [8].

Table 17 - ATX Power Supply Minimum Current Requirements

Voltage (V)	Minimum Current
+12	2 Amps
+5	3 Amps
+3.3	0.3 Amps
-5	0.1 Amps
-12	0.1 Amps

If the minimum current requirements are not met, the power supply will not function. The Pixel Router was not designed to accept power at all of these voltages, thus external resistors capable of handling high currents were applied to the power supply outputs to meet the current requirements. The circuit diagram for this is shown below.

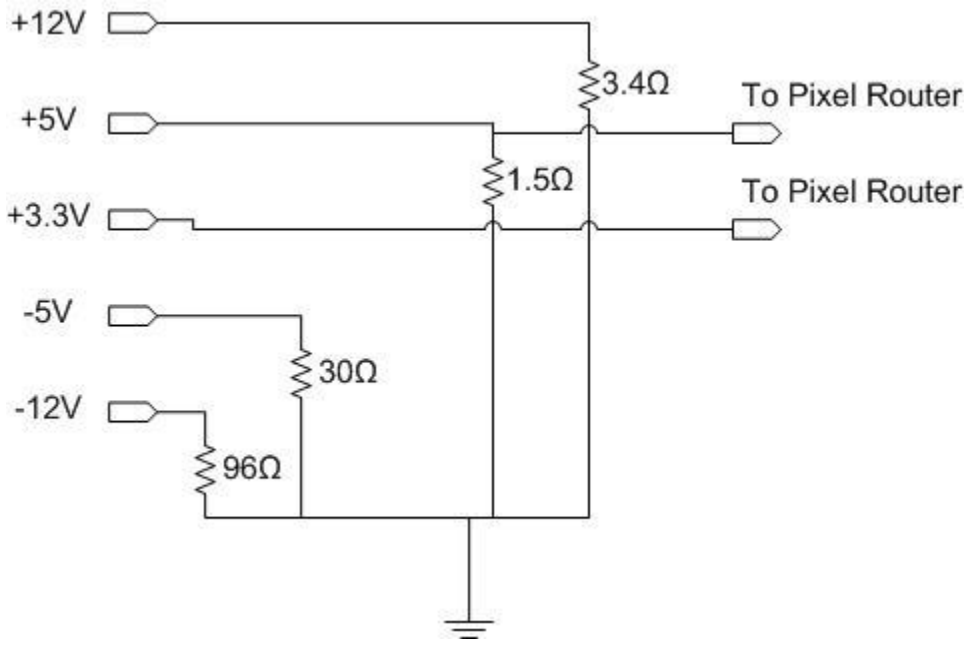

Figure 17 - ATX Power Supply Power Dissipation Schematic

This design is extremely wasteful, as over 60W of power is essentially dissipated as heat. In fact, for safety reasons the 3.4Ω and 1.5Ω resistors, which constitute the bulk of the dissipated power, are attached directly to a heat sink which has a fan blowing over the plates. Temperature measurements for these resistors were taken with the heat sink applied, and are listed below with the 1.5Ω resistor labeled R1 and the 3.4Ω resistor labeled R2.

Table 18 - Heat Characteristics of Power Dissipation Resistors

Time (min)	R1 (°C)	R2(°C)
0	24.1	24.1
1	34.4	55
2	38	64
3	47	73
4	48	80.5
5	50.5	83.5
6	46.5	86
7	48.5	78
8	51.5	81.5
9	50	81
10	50	85

Warning labels would need to be placed on or near the resistors stating that they should not be touched, as they get quite hot. There is, however, no risk of fire or other damage due to their temperature. Future iterations of packaging for the Pixel Router will include either a power supply designed specifically to meet its requirements or an off the shelf supply with fewer or no minimum current requirements. This will reduce or eliminate the need for these resistors, which have the potential to be hazardous to end users.

FCC Certification

For products to be sold within the US, they must undergo testing to ensure they meet certain safety and regulatory requirements. A requirement for electronic products is that they pass FCC conducted and radiated emissions testing that verifies the device is not unintentionally broadcasting electromagnetic signals at certain frequencies and power levels over the air or through power lines. This helps prevent noise and interference between devices that could limit their effectiveness or capability. The requirements for devices classified as “Unintentional Radiators”, of which the Pixel Router is one, is specified in Part 15, Subpart B of the FCC regulations. Devices under these regulations fall under two categories for which two sets of standards exist. The first category is “Class A Digital Devices”, which constitute devices “marketed for use in a commercial, industrial, or business environment” [9]. The second category is “Class B Digital Devices”, consisting of devices “marketed for use in a residential environment”.

The Pixel Router is intended for commercial purposes, therefore it is required to meet Class A standards, which have tighter restrictions on conducted and radiated emissions limits. These limits, taken from the FCC Part 15 guidelines [9], are listed in the tables below. Note that for radiated emissions the units are converted from µV/m to dBµV/m for consistency with actual measurements taken during compliance testing for the Pixel Router.

Table 19 - FCC Part 15 B Class A Radiated Emissions Limits [9]

Frequency of Emission (MHz)	Field Strength (dBµV/m)
30-88	39
88-216	43.5
216-960	46.4
Above 960	49.5

Table 20 - FCC Part 15 B Class A Conducted Emissions Limits [9]

Frequency of Emission (MHz)	Conducted Limit (dBµV)	
	Quasi-peak	Average
0.15-0.5	79	66
0.5-30	73	60

To ensure compliance with these standards and obtain legal marketability, the Pixel Router underwent conducted and radiated emissions testing. The Pixel Router is capable of operating in two modes: Pass-through Mode, in which the input sources are directly mapped to the video outputs and no transformation is applied, and blended mode, in which the LUT-based image transformation is applied. Changes in operation of a device can affect its emissions characteristics due to changes in which signal traces are active and at which frequency, thus the Pixel Router was tested in both modes of operation. Intertek Testing Services NA, Inc. performed the compliance testing in their 10m semi-anechoic chamber. This chamber acts as a tightly sealed Faraday cage which lets very little electromagnetic radiation enter or escape and minimizes electromagnetic reflection on interior surfaces, ensuring results that are unperturbed by outside interference.

The Pixel Router was non-compliant for both conducted and radiated emissions tests on the first pass for each. Some minor modifications were able reduce the emissions levels in both cases, however, and provide compliant results. The initial non-compliant frequency scan for the radiated emissions test is shown below. This test was conducted in Pass-through Mode. In the plot, the red line indicates the compliance threshold, the blue line indicates the emission values for vertically polarized waves, and the green line for horizontally polarized waves.

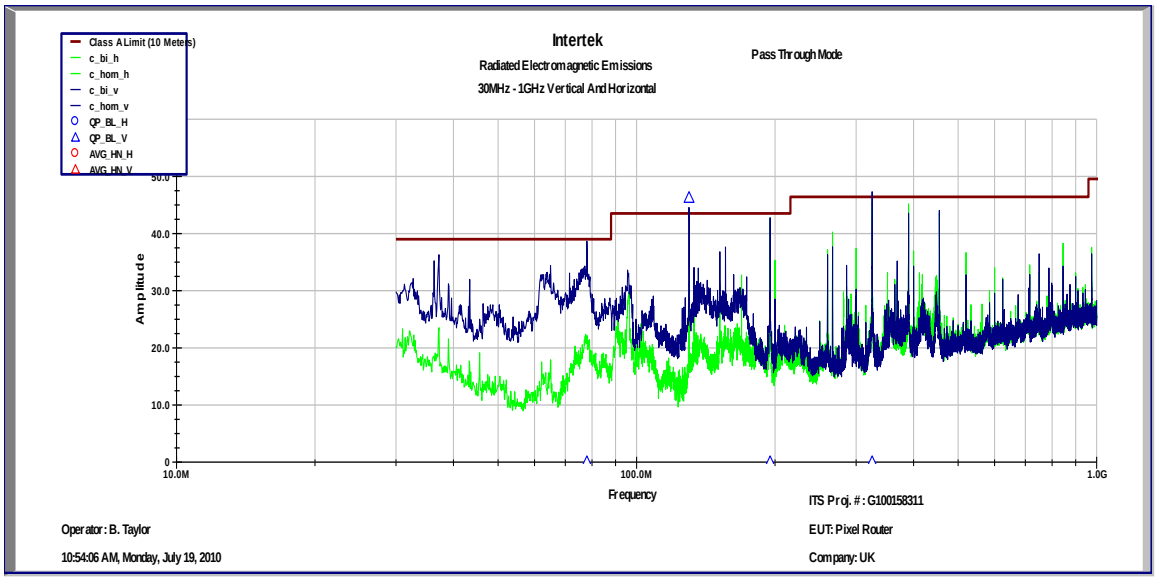

Figure 18 - Initial Radiated Emissions Scan

In the plot above, the blue triangle above the red line indicates a frequency that was closely examined for compliance due to the proximity of the peak in the initial scan to the compliancy level. In this case, the emissions at that frequency (130.04 MHz) was found to be non-compliant with a value of 46.23 dBµV/m versus a compliancy limit of 43.5 dBµV/m. It was found that the cause of this non-compliance was poor connection and ground of several of the HDMI cables on the rear of the device. The custom HDMI connection port that was designed to allow 8 HDMI cables to be connected was not designed with emissions compliance in mind. Thus, the male and female cable ends would not always make a tight connection, allowing radiation to escape. Two modifications were done to allow for FCC compliance. The first was to support the cables near the connection point to prevent sagging. This allowed the cables to enter the female end at an angle closer to 90 degrees, creating a more secure connection. The other modification was the application of an “RF gasket” along the top edge of the connection points. This gasket is essentially conductive tape which creates a better connection between the ground shield on the HDMI cables and the chassis ground of the device. An image of the rear connection with these modifications is shown below.

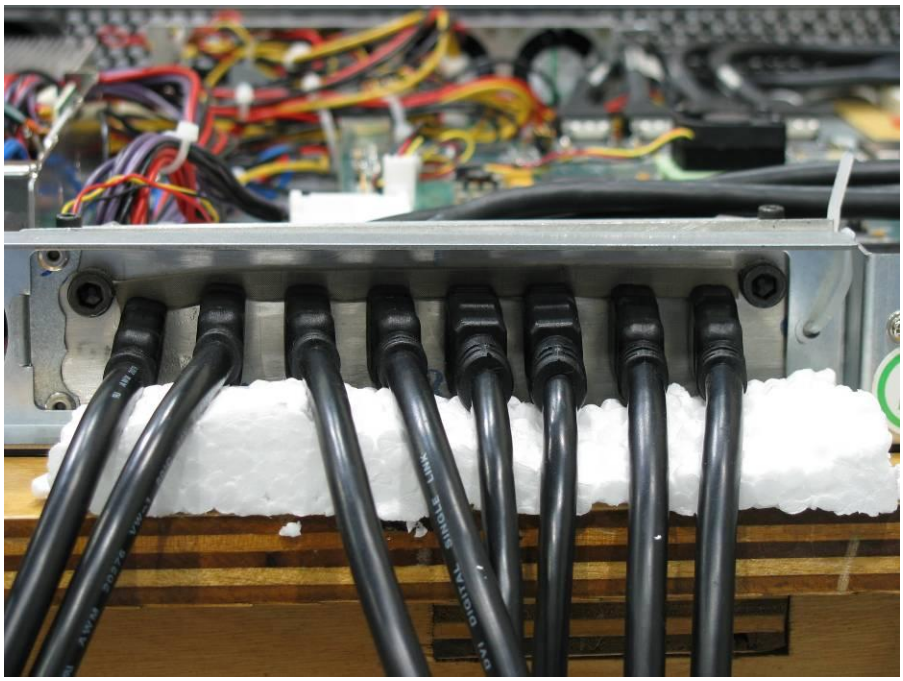

Figure 19 - Support for HDMI Cables

In the above image, the RF gasket is the darker strip along the top of the cables, and the cable support is the white Styrofoam underneath the cables. Minor modifications such as these are allowed to be included to produce compliant results, provided that sufficient steps are taken in the final product to address the issues alleviated by the modifications. In a production version of the Pixel Router, these issues would be addressed by redesigning the HDMI connection port using standard HDMI jacks rather than male-female extension cables. In addition to reducing the emissions spectrum, such a design change would also give the Pixel Router the look and feel of a professionally designed commercial device. The chart below shows the results of emissions testing after these modifications, which produced compliant results.

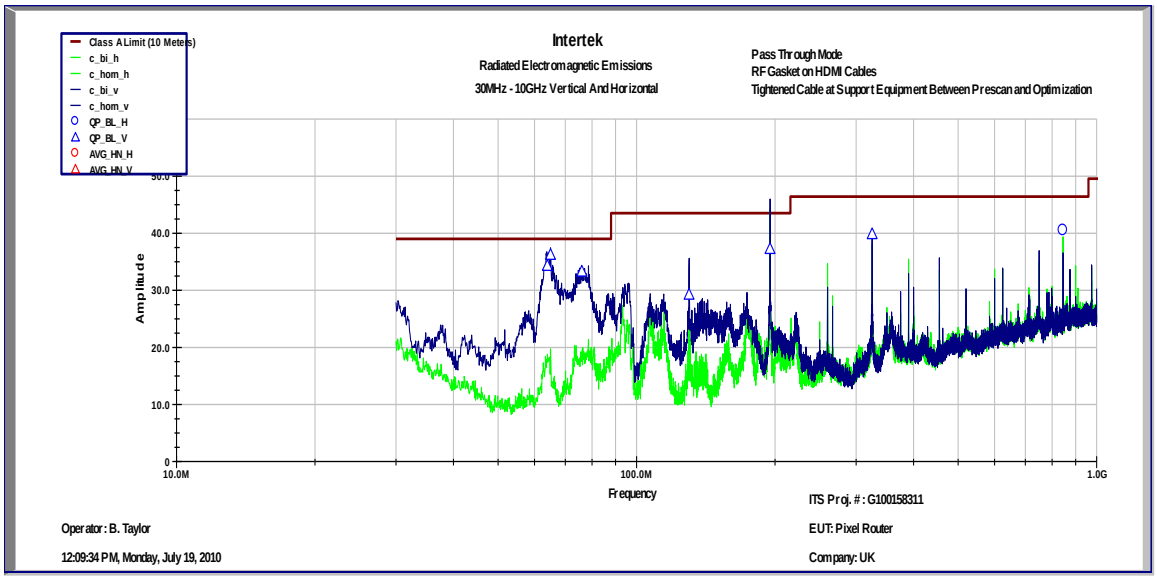

Figure 20 - Compliant Radiated Emissions Scan

In the chart above, there is a blue spike at 195 MHz which appears to be non-compliant as it crosses the red line. However, upon close examination around that frequency, it was determined that the actual reading was 37.23 dBµV/m which is less than the compliant value at that frequency of 43.5 dBµV/m. Thus, the radiated emissions were determined to be within compliant limits.

The conducted emissions for the Pixel Router were also found to be non-compliant upon first scan. The chart below shows the results of this initial scan. In this test, the emissions are measured on the two power supply lines ("phase" and "neutral" for a 3-pronged power outlet) relative to ground, which are referred to as Line 1 and Line 2. The green line in the chart below represents Line 1, and the blue line represents Line 2. Additionally, there are two standards for compliance, Quasi-peak and Average. The Quasi-peak compliancy levels are shown by the black line on the chart, and the Average compliancy levels are shown by the red line.

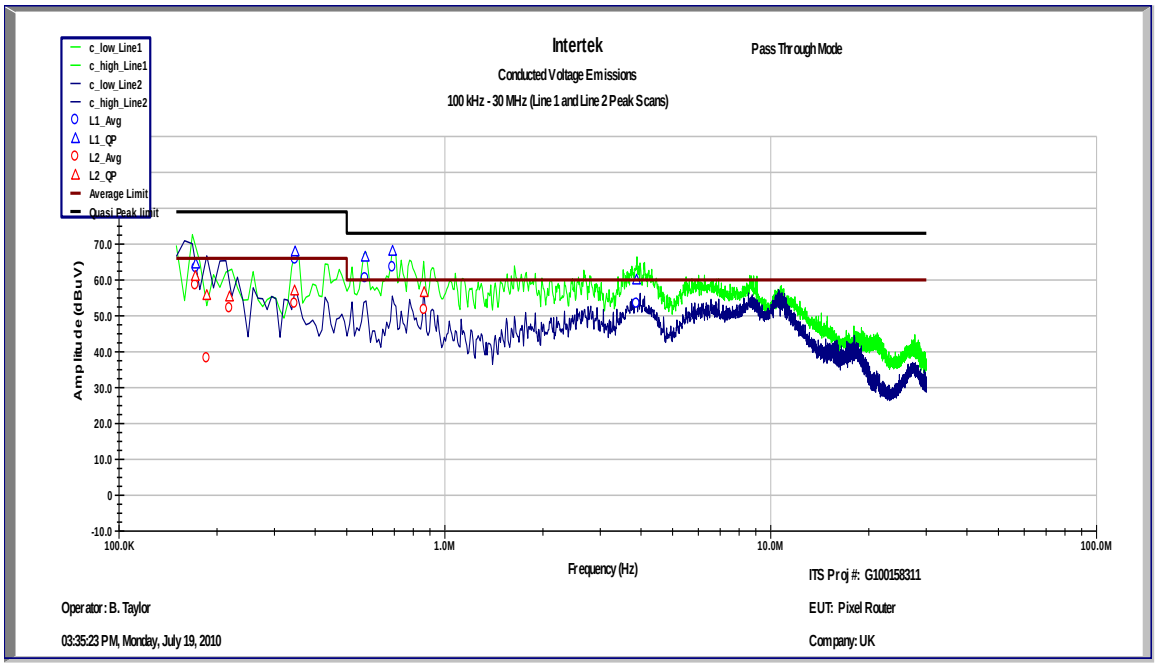

Figure 21 - Initial Conducted Emissions Scan

In the above scan, while the quasi-peak values are compliant, the average values on Line 1 are above compliancy levels resulting in a non-compliant test. Compliancy for conducted emissions is often related to the power supply of a device and how much filtering circuitry is in place to minimize emissions. The off-the-shelf power supply chosen for the Pixel Router was operating in a non-compliant mode in this case. To correct this non-compliancy, an in-line power filter (part number 15EF1F) was placed between the power connection and the power supply which acted as a low-pass filter to reduce the high frequency noise that was being emitted. An image of this modification is shown below.

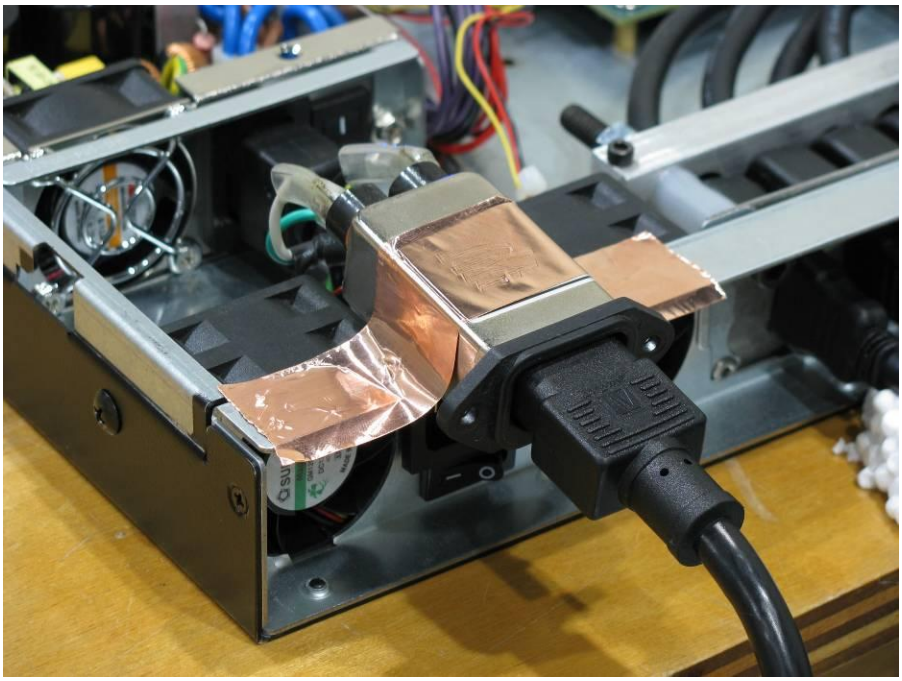

Figure 22 - In-line Power Filter

In the above image, the in-line power filter is attached to the case of the Pixel Router with conductive copper tape, which ensured a connection to chassis ground. The circuitry of the filter is shown in the schematic below, which is taken from the datasheet of the device [11]. The component values, listed on the outside label of the device, are as follows: L = 22.27H, C = 0.047µF (x2), R = 1.5MΩ.

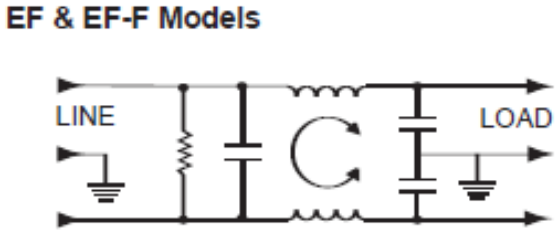

Figure 23 - In-line Power Filter Schematic [11]

This in-line filter significantly reduced the conducted emissions levels of the Pixel Router. The results of the second conducted emissions scan are shown below.

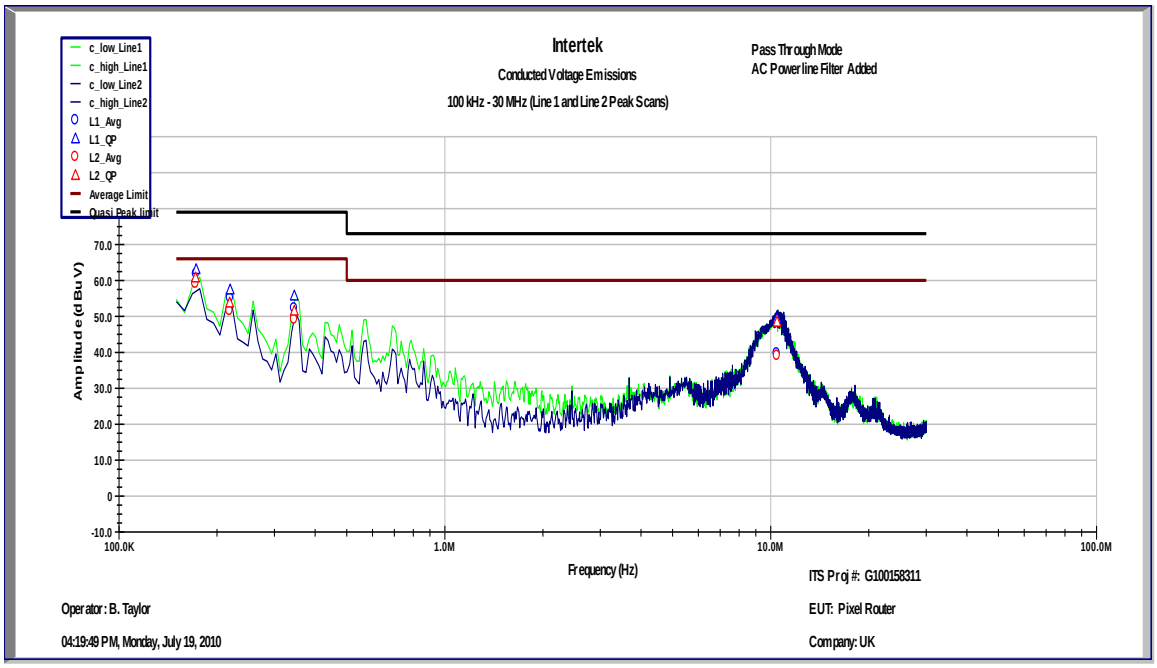

Figure 24 - Compliant Conducted Emissions Scan

This chart shows the conducted emissions values well below compliance limits in both the average and quasi-peak measurements. In a production version of the Pixel Router a different power supply would likely be chosen. This would not only likely fix the conducted emissions non-compliance, but also address the need to waste power due to minimum current requirements on different voltage levels. If a different power supply was chosen, conducted and radiated emissions tests would need to be performed again to ensure compliance. If the same power supply was to be used, an in-line power filter such as the one used in testing would need to be installed to remain in compliance.

Section 7: Business Strategy

Potential Markets

The Pixel Router was designed to be a commercially viable product for performing image blending and warping for displays with multiple projectors. With the improvements made to the image quality and frame rate, and obtaining the required federal certifications, commercial opportunities for the product can begin to be explored. While the Pixel Router hardware could be useful in applications outside of multi-projector edge blending, for the purposes of this project only edge-blended display applications will be considered.

A number of industries are currently making use of multi-projector displays for various purposes. These industries can be broadly divided into two categories: military/government agencies and private industry. The two primary needs for multi-projector displays in the military sector are in troop training and simulation displays and command and control centers. In FY2011, the US Army allocated $118M toward command and control procurement and $354M toward troop training and simulation systems procurement [12]. Additionally, $185M was allocated by the Army for research and development in command and control systems and $35M was allocated for research and development of Next Generation Training and Simulation systems [13]. The total funds allocated in all of these categories amounts to $693M, which shows a strong demand by the Army for these types of systems. Ultra-high resolution displays that can be rapidly set up and calibrated help meet the needs of the US military in both of these categories. Such systems can provide a large screen to display content such as high-resolution images, video feeds, and other relevant information about a mission. Additionally, the flexibility of arbitrary projector calibration software allows for the creation of immersive training and simulation environments that can provide a sense of reality to the soldier. The Pixel Router hardware is well positioned to meet these needs by allowing multiple video inputs of arbitrary content rather than requiring source code modification like many multi-projector software solutions. Additionally, because the Pixel Router is FPGA-based hardware that does not run an operating system or connect to an IP network, it is highly reliable, requires no “boot-up” time, and is not prone to IP-based network attacks, making it an ideal solution for mission critical or information sensitive displays.

In-roads have been made in establishing the Pixel Router as a viable option for multi-projector displays for the US military. In 2008, a Phase II STTR sponsored by the Navy was awarded to the University of Kentucky for development and commercialization of this device. STTR (Small Business Technology Transfer) grants are awarded as a partnership between a research institution and a small private company with the intent of assisting both entities in commercializing research. Upon completion of a Phase II contract, the product developed under the STTR is eligible for a Phase III award, which is an open-ended contract for production and procurement. The STTR awarded by the

Navy to the University of Kentucky was sponsored by the Naval Air Command for the development of portable immersive training systems. The goal of this research is to develop shipboard flight simulators for mission rehearsal and training. Through the completion of the Phase II project and research, the Pixel Router will be well positioned for future potential contracts with the Navy or other branches of the armed forces.

The private sector also presents a market opportunity for the Pixel Router and large-format, high-resolution displays. Such displays could be useful in engineering design visualization, teleconferencing and telepresence, and command and control centers. As in the military sector, the reliability, security, and rapid deployability of the Pixel Router specific niche markets within private industry. One of these markets could be in the creation of a portable telepresence display to enable traveling executives the ability to conduct meetings while away from the conference room. Another example market is trade show booth displays, where the largest, flashiest booth often draws the most attention. In such displays, the portability and ease of set up is an extremely important factor. While the private sector does present opportunities for the Pixel Router, the primary market focus should be on the government and military sector where large market size and lower barriers to entry allow for easier adoption and rapid expansion of business.

Cost of Materials

It outside of the scope of this document to examine the specific financial conjectures of any potential company that could be formed to develop and market the Pixel Router. However, to determine the viability of the product, the cost to produce the device is examined here. Below is a table showing the costs associated with the most recent production of hardware, which occurred in mid 2008.

Table 21 – Pixel Router Cost of Materials and Production [14]

Item	Cost per Board
Surface Mount Components	$1,302
Case	$138
Power Supply	$82
PCB Fabrication	$335
Assembly	$492
Miscellaneous Manufacturer Expenses	$190
TOTAL	**$2,539**

These prices were quoted for a very small run of production intended only for research and testing, not for sale. Should the need arise for larger volume production, it is believed that the cost of materials and production could be significantly reduced due to bulk order discounts. The cost of producing the Pixel Router makes it unlikely that the device could ever be practical for the consumer market, however the commercial and

military markets would easily be able to bear the cost for such a device, even with a significant profit margin.

Section 8: Conclusion and Future Work

Significant steps toward commercializing the Pixel Router have been taken by improving performance in frame rate and image quality, as well as packaging the device and certifying it for compliance with the FCC. An effective cache system was designed that allowed for the implementation of bilinear interpolation at frame rates of at least 20 fps for up to 5 degrees of image rotation. The maximum frame rate achieved in simulation was measured at over 33 fps for 1 degree of rotation on the image, and in practice the maximum frame rate measured was 30 fps for 1 degree of rotation. The previous design of the Pixel Router, which used nearest neighbor interpolation, only performed at 20 fps. Thus, the cache system allowed for a 50% improvement in frame rate while processing 4 times as many pixels to perform bilinear interpolation. The robust, portable design of the packaging for the Pixel Router and obtaining FCC certification for the device allow it to be marketed and legally sold in the US.

Significant work remains to be done, however, to move the Pixel Router toward true marketability. A frame rate of 20 to 30 fps, while acceptable in some applications, is below most users' expectations of 60 fps, which is standard on most video devices. One way to address this problem is to increase the clock speed of the processor and memory on the device. Currently both components are operating at 133MHz. Modern DDR3 SDRAM devices are capable of operating at over 1333MHZ, which could improve the frame rate capability on the Pixel Router by a factor of 10. Another topic of future work for the Pixel Router is to improve the ease of use of the device by calculating the projector blending and warping masks on the device, rather than in an offline step. This would allow the Pixel Router to act as a completely stand-alone device, whereas currently a PC is required for the calculation of the LUTs. This would require implementing a microcontroller on Pixel Router that is capable of performing these calculations. Upgrading the RAM hardware and on-board LUT generation would transform the Pixel Router into a truly compelling product with the potential to achieve broad adoption across a variety of markets.

Appendix A: LUT Generation Code

The following code, written in C++, is used to generate the LUTs used by the Pixel Router.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<imdebug.h>
#include<iostream>
#include<math.h>
#include "CacheFunctions.h"


using namespace std;

const int ROTATION_ANGLE = 45;

/*////////////////////////////////////////////////////////
//////////////////
Cache Fill Description

The cache is filled in the following order:
Bank 0, Line 0
Bank 1, Line 0
Bank 0, Line 1

Bank 1, Line 1
Bank 0, Line 2
...
Bank 0, Line 7
Bank 1, Line 7

There are 2 cache buffers (containing 2 cache banks each)
that "ping-pong".
As one cache buffer is being loaded with data from DRAM,
the other cache
buffer is being drained and providing data to the output.

To conserve memory bandwidth, each cache line is only
loaded with the number
of valid pixels for that fill.  Valid pixels can be any
pixel used for bilinear
interpolation.
```

```
Each cacheLine contains a "numPixels" field defining the
number of pixels that
have been loaded in to it.

The Prefetch LUT will be written such that cache lines
containing the same
row will be written consecutively to minimize the overhead
penalty for
opening a new cache row.

/*////////////////////////////////////////////////////////
//////////////////
int round(float);
char * convert(long int);

void writelut(void);
void get_input_pixel(float&, float&);
int get_fractional_bits(float, bool&);
//void DecodePixel(Pixel&);

//void EncodePixel(Pixel&);
void WritePrefetchLUT(int, cacheStart, int&);
void WritePerPixelLUT(int, int, int, int, int, int, int);


float* Rotate(int, int, int);
//bool CheckNextPixel(Pixel, cacheStart);

bool PIP = false;  //Picture In Picture
int PIP_U_OFFSET = 800;
int PIP_V_OFFSET = 450;
int PIP_WIDTH = 320;
int PIP_HEIGHT = 180;


int salpha=0;
long int *lut;
int *fracBits;
unsigned int *alpha;

int perPixelEntries = 0;

int
no_of_proj,novertices,notri,minpixu,minpixv,maxpixu,maxpixv
;
float maxu=-1e10, maxv=-1e10, minu=1e10, minv=1e10, urange,
vrange;
```

```
float miniu,miniv,maxiu,maxiv,uprime,vprime;
float *ucoord, *vcoord,*mesh;


int main(int argc, char * argv[]) {
     char *filez,*alphaz,strfile[2],blinz[20],*blinzi;
     char alphaa[20];
     char perPixName[20], prefetchName[20];
     int filecnt=1;
     int pip_int = 0;
     if(argc == 12)
     {
          IWIDTH = atoi(argv[1]);
          IHEIGHT = atoi(argv[2]);
          OWIDTH = atoi(argv[3]);
          OHEIGHT = atoi(argv[4]);
          ICHANNELS = atoi(argv[5]);
          OCHANNELS = atoi(argv[6]);
          pip_int = atoi(argv[7]);
          if(pip_int == 1)
               PIP = true;
          else
               PIP = false;
          PIP_U_OFFSET = atoi(argv[8]);
          PIP_V_OFFSET = atoi(argv[9]);
          PIP_WIDTH = atoi(argv[10]);
          PIP_HEIGHT = atoi(argv[11]);
     }
     else if(argc == 7)
     {
          IWIDTH = atoi(argv[1]);
          IHEIGHT = atoi(argv[2]);
          OWIDTH = atoi(argv[3]);
          OHEIGHT = atoi(argv[4]);
          ICHANNELS = atoi(argv[5]);
          OCHANNELS = atoi(argv[6]);
          PIP = false;
     }
     else if(argc == 5)
     {
          IWIDTH = atoi(argv[1]);
          IHEIGHT = atoi(argv[2]);
          ICHANNELS = atoi(argv[3]);
          OCHANNELS = atoi(argv[4]);
          PIP = false;
     }
     else if(argc == 4)
```

```
	{
		IWIDTH = atoi(argv[1]);
		IHEIGHT = atoi(argv[2]);
		ICHANNELS = atoi(argv[3]);
		PIP = false;
	}
	else
		printf("Correct Usage: finalpolate.exe [IWIDTH]
[IHEIGHT] [OWIDTH] [OHEIGHT] [ICHANNELS] \n[PIP]
[PIP_U_OFFSET] [PIP_V_OFFSET] [PIP_WIDTH] [PIP_HEIGHT]");


	char a[30],b[10];
	int i,j,temp,k;
	float **vertices; // changing the number of vertices
	int **triangles; // changing the number of triangles
	alphaz=&alphaa[0];
	blinzi=&blinz[0];
	float tx1,ty1,tx2,ty2,tx3,ty3,tu1,tv1,tu2,tv2,tu3,tv3;
	int tt1,tt2,tt3;
	char *try1,lutv[33];
	long int cnt=0;
	lut=new long int [OWIDTH*OHEIGHT];
	fracBits = new int [OWIDTH*OHEIGHT]; //Fractional Bits
for Bilinear Interpolation
	alpha=new unsigned int[OWIDTH*OHEIGHT];
	ucoord= new float[OWIDTH*OHEIGHT];
	vcoord= new float[OWIDTH*OHEIGHT];
	//mesh=new float[1024*768];
	//flex(); // reading flex file values
	//Binary_flex(); // Reading basic info (number of
projectors, inside BBox)
//	while(filecnt<no_of_proj) {
		for(j=0;j<OHEIGHT;j++) {
	//initializing lut,alpha,u and v
			for(i=0;i<OWIDTH;i++) {
				ucoord[j*OWIDTH+i]=-1;
				lut[j*OWIDTH+i]=-1;
				vcoord[j*OWIDTH+i]=-1;
				alpha[j*OWIDTH+i]=-1;
		//		mesh[j*1024+i]=-1;
			}
		}


		strcpy(alphaa,"lut");
		strcpy(blinz,"cache_lut");
```

```
            strcpy(perPixName,"perPixelTest");
            strcpy(prefetchName,"prefetchTest");
            _itoa(filecnt,strfile,10);
            strcat(alphaa,strfile);
            strcat(blinz,strfile);
            strcat(perPixName,strfile);
            strcat(prefetchName,strfile);
            strcat(alphaa,".txt");
            strcat(blinz,".txt");
            strcat(perPixName,".csv");
            strcat(prefetchName,".csv");
            printf("\n%s",alphaz);
            f2=fopen(alphaz,"wb+");   // opening the per-
pixel lut  ****Changed to w+ from wb+ to support testing
            f3=fopen(blinzi,"wb+");   // opening the
prefetching lut ****Changed to w+ from wb+ to support
testing
            perPix=fopen(perPixName,"w+");
            preFetch=fopen(prefetchName,"w+");
//          lutFile.open(alphaz, ios::out | ios::binary);
//          fprintf(f2,"%d %d %d\n",filecnt,OWIDTH,OHEIGHT);
            //Write header for LUTs in binary format
////////////////////////////////////////////////////////////
////////////////////////////////////
//          Changes to support Ben in FPGA testing
////////////////////////////////////////////////////////////
////////////////////////////////////
            int* p;
            p = &filecnt;
            fwrite(p,sizeof(int),1,f2);
            p = &OWIDTH;
            fwrite(p,sizeof(int),1,f2);
            p = &OHEIGHT;
            fwrite(p,sizeof(int),1,f2);
//          int dummy = 55;
//          p = &dummy;
//          fwrite(p,sizeof(int),1,f2);
            for(int i = 0; i < 3; i++)
                 fwrite(p,sizeof(int),1,f3);


////////////////////////////////////////////////////////////
////////////////////////////////////
            //fprintf(f3,"\n");//Header space at the top,
placeholder for size information
            strcpy(alphaa,"test");
            _itoa(filecnt,strfile,10);
```

```
            strcat(alphaa,strfile);
            strcat(alphaa,".csv");
            printf("\n%s",alphaz);

            writelut();
            filecnt=filecnt+1;
//          fclose(fp_binary);
            fclose(preFetch);
            fclose(perPix);
//    }
      printf("\nFinished\n");
      getch();
      delete(lut);
      delete(alpha);
      delete [] ucoord;
      delete [] vcoord;
      delete(vertices);
      delete(triangles);
      return(0);
}



int round(float unrndval) {
      int temp1=unrndval;
      if(unrndval>=(temp1+0.5)) {
            temp1=temp1+1;
      }
      return(temp1);
}


char * convert(long int lutval) {
      char
a[11],b[33],a1[11],b1[33],c[33]="000000000",c1[9],d[5];
      char *ret;
      int k,i,j=0;
//    itoa(col,a,2);
//    itoa(row,b,2);
      itoa(lutval,b1,2);
      for(k=0;k<(32-strlen(b1));k++) {
            c[k]='0';
      }
      c[32-strlen(b1)]=NULL;
      strcat(c,b1);
      ret=&c[0];
      return(ret);
```

```
}
void writelut (void) {

     int i,j,k,trow,tcol,talpha=0,index=0;
     int ufrac, vfrac;
     int uprime_int;
     int vprime_int;
     char *lutval,lutv[33];
     unsigned char a1,a2,a3,a4;
     unsigned int check=0;
     long int store=0,offset;
     FILE* prefetchTest;
     FILE* perPixTest;

     float *c;
     int angle = ROTATION_ANGLE;
     bool increment = false;

     char preName [17] = "prefetchTest.csv";
     char perPixName [17] = "perPixText.csv";

     prefetchTest = fopen(preName, "w");
     perPixTest = fopen(perPixName, "w");

     //First pass FOR loops - Get input addresses and store
them in an array
     for(j=0;j<OHEIGHT; j++){  //change
          for(i=0;i<OWIDTH;i++) {
               c = Rotate(i, j, angle);
               fprintf(perPixTest,"%d,%d,%4.2f,%4.2f\n",j,
i, c[1],c[0]);
               tcol = c[0];
               trow = c[1];
               talpha = 255;
               index = 0;
               ufrac = get_fractional_bits(c[0],
increment);
               //REMOVE FOR TESTING
               //if(increment)
               //   tcol++;
               vfrac = get_fractional_bits(c[1],
increment);
               //REMOVE FOR TESTING
               //if(increment)
               //   trow++;

               //TEST CODE//
```

```
//              ufrac = 0;
//              vfrac = 0;
                //tcol = tcol - 2;
                if (tcol < 0 || tcol >=IWIDTH)
                {
                     talpha = 0;
                }
                //trow--;
                if (trow < 0 || trow >=IHEIGHT)
                {
                     talpha = 0;
                }


                fracBits[j*OWIDTH+i] =
((vfrac&7)<<3)|(ufrac&7);

                offset=trow*IWIDTH+tcol;


                //TEST MODE:  JUST USE PIXEL 0
/*              offset = 0;
                talpha = 255;
                index = 0;
                fracBits[j*OWIDTH+i] = 0;

*/


                if ((talpha&255) == 0)
                {
                     store = 0x80000000;
                     lut[(/*OHEIGHT-1-*/j)*OWIDTH+i] =
store;
                }
                else
                {
                     store=(0<<31) | ((talpha&255)<<23) |
((index&3)<<21) | ((offset&0x001fffff));
                     lut[(/*OHEIGHT-1-*/j)*OWIDTH+i] =
store;
                }
           }
     }
          //Initial pass FOR loops

     //Simulated cache values
     cacheStart cacheBuffer;
```

```
      cacheStart cacheWriteBuffer;

      sortedCache cacheSorted;

      InitializeCache(cacheBuffer);
      InitializeCache(cacheWriteBuffer);
      InitializeSortedCache(cacheSorted);

      bool inCache = false; //Pixel in cache?
      bool firstEntry = true;
      bool nearestNeighbor = false;
      long lutOffset;
      int lutAlpha;
      int lutFrac;
      int channelID;
      int lutIgnore;
      int command;
      int bank0Column = 0;
      int bank1Column = 0;
      int prefetchSize=0;

      int bank0Line = 0;
      int bank1Line = 0;

      //Variables for delayed per-pixel write
      //This is done so that a command of '1' shows up on
the last valid cache hit
      int prevLutAlpha;
      int prevLutFrac;
      int prevCommand;
      int prevBank0Column;
      int prevBank1Column;
      int prevBank1Line;
      int prevBank0Line;

      int lastRowOpened=0;

      int refreshCommands = 0;


      Pixel currentPixel;
      Pixel ptrPixel;


      //Second pass FOR loops - Determine per-pixel and
prefetching LUT values, write to file
      for(int r = 0; r < OHEIGHT; r++)
```

```
     {
          for(int c = 0; c < OWIDTH; c++)
          {
               //Store row, column data in Pixel for easy
portability
               ptrPixel.row = r;
               ptrPixel.column = c;
               EncodePixel(ptrPixel);

               //Determine if pixel is valid
               lutIgnore = DecodeIgnore(lut[r*OWIDTH+c]);
               if (lutIgnore != 1)
               {
                    //Get the current pixel address and
data
                    currentPixel.address =
DecodeOffset(lut[r*OWIDTH+c]);
                    DecodeInputPixel(currentPixel);

                    channelID = DecodeCID(lut[r*OWIDTH+c]);
                    lutAlpha =
DecodeAlpha(lut[r*OWIDTH+c]);
                    lutFrac = fracBits[r*OWIDTH+c];

                    //CheckCache returns true unless cache
is full
                    //If room exists in the cache, a pixel
is placed there in CheckCache
                    inCache = CheckCache(currentPixel,
ptrPixel, cacheBuffer, channelID, bank0Column, bank1Column,
                                                  command,
nearestNeighbor, bank0Line, bank1Line);

                    //Set fractional bits to 0 for nearest
neighbor
                    if(nearestNeighbor)
                         lutFrac = 0;


                    if(!inCache)
                    {
                         //Sort the cache for writing;
                         SortCache(cacheBuffer,
cacheSorted);

                         //Write the cache
                         WriteCache(cacheSorted);
```

```
                              //16 entries are written for every
cache miss.
                              prefetchSize += 2*CACHE_LINES;


                              if(!firstEntry)
                              {
                                   WritePerPixelLUT(1,
prevBank1Line, prevBank0Line, prevBank1Column,
prevBank0Column,
                                                  prevLutAlpha,
prevLutFrac);
                                   perPixelEntries++;
                                   refreshCommands++;
                              }


                              //Reset cache, assume that none of
the data can be reused.
                              InitializeCache(cacheBuffer);

     InitializeSortedCache(cacheSorted);

                              //Place pixel in cache

                              inCache = CheckCache(currentPixel,
ptrPixel, cacheBuffer, channelID, bank0Column, bank1Column,
                                   command, nearestNeighbor,
bank0Line, bank1Line);

                              prevCommand = 0;//1 is already
written, don't write it again next time
                              prevBank1Line = bank1Line;
                              prevBank0Line = bank0Line;
                              prevBank1Column = bank1Column;
                              prevBank0Column = bank0Column;
                              prevLutAlpha = lutAlpha;
                              prevLutFrac = lutFrac;



                         }//!In Cache If
                         else
                         {
                         //Write Per Pixel LUT
                              if(!firstEntry)
```

```
                              {
                                   WritePerPixelLUT(prevCommand,
prevBank1Line, prevBank0Line, prevBank1Column,
                                        prevBank0Column,
prevLutAlpha, prevLutFrac);
                                   perPixelEntries++;
                                   if(prevCommand==1)
                                        refreshCommands++;
                              }
                              else //Debug first entry
                                   cout<<"First entry:
"<<command<<", "<<bank1Line<<", "<<bank0Line<<", "<<
bank1Column<<
                                        ", "<<bank0Column<<",
"<<lutAlpha<<", "<<lutFrac<<endl;

                              //pixelsToLoad++;

                              prevCommand = command;
                              prevBank1Line = bank1Line;
                              prevBank0Line = bank0Line;
                              prevBank1Column = bank1Column;
                              prevBank0Column = bank0Column;
                              prevLutAlpha = lutAlpha;
                              prevLutFrac = lutFrac;
                         }

                    }//Ignore If
                    else
                    {
                         //Ignore
                         if(!firstEntry)
                         {
                              WritePerPixelLUT(prevCommand,
prevBank1Line, prevBank0Line, prevBank1Column,
                                   prevBank0Column,
prevLutAlpha, prevLutFrac);
                              perPixelEntries++;
                              if(prevCommand==1)
                                   refreshCommands++;
                         }

                         prevCommand = 0;
                         prevBank1Line = 0;
                         prevBank0Line = 0;
                         prevBank1Column = 0;
                         prevBank0Column = 0;
```

```
                    prevLutAlpha = 0;
                    prevLutFrac = 0;
               }

               //This makes sure that nothing is written
for the first value so that we don't have extra data
               if (firstEntry)
                    {
                         firstEntry = false;
                    }

          }//Per-pixel FOR loops
     }
     //Finished FOR loops, write last per-pixel entry;
     //Make sure a '1' is written as the command
     WritePerPixelLUT(1, prevBank1Line, prevBank0Line,
prevBank1Column,
                              prevBank0Column,
prevLutAlpha, prevLutFrac);
     perPixelEntries++;
     refreshCommands++;

     cout<<"Per Pixel Entries:  "<<perPixelEntries<<endl;
     cout<<"Cache Refresh Commands:
"<<refreshCommands<<endl;
     //Sort the cache for writing;
     SortCache(cacheBuffer, cacheSorted);

     //Write the cache
     WriteCache(cacheSorted);

     //16 entries are written for every cache miss.
     prefetchSize += 2*CACHE_LINES;

     //Write Prefetch Size Info at top of file
////////////////////////////////////////////////////////////
////////////////////////////////////
//   Changes to support Ben in FPGA testing
////////////////////////////////////////////////////////////
////////////////////////////////////

     fseek(f3,0,0);
     prefetchSize *= 4; //Prefetch Size in Bytes
     int*p;
     int dummy = 99;
     p = &dummy;
     fwrite(p,sizeof(int),1,f3);
```

```
	p = &prefetchSize;
	fwrite(p,sizeof(int),1,f3);
	dummy = 0;
	p = &dummy;
	fwrite(p,sizeof(int),1,f3);

	fclose(f3);
////////////////////////////////////////////////////////////
////////////////////////////////////////
	printf("\n%d\n", prefetchSize/4);
}


void get_input_pixel (float& vcoord, float& ucoord)
{
	//u: WIDTH
	//v: HEIGHT
	float vcoord_norm = vcoord;
	float ucoord_norm = ucoord;
	float INWIDTH = IWIDTH/1.0;
	float INHEIGHT = IHEIGHT/1.0;
	float OUTWIDTH = OWIDTH/1.0;
	float OUTHEIGHT = OHEIGHT/1.0;
	unsigned int vf = 0, uf = 0;

	//Row, column projector index
	if (vcoord_norm > 0.5)
		vf = 1;
	if (ucoord_norm > 0.5)
		uf = 1;

	//One Input channel
	if (ICHANNELS == 1)
	{
		//Channel 1 across all 4 projectors
		ucoord = ucoord_norm*(INWIDTH);
		vcoord = vcoord_norm*(INHEIGHT);
//		ucoord = round(ucoord_norm*(INWIDTH/2) +
uf*(INWIDTH/2));
//		vcoord = round(vcoord_norm*(INHEIGHT/2) +
vf*(INHEIGHT/2));
	}
	//Two Input Channels
	else if (ICHANNELS == 2)
	{
		//Picture In Picture enabled
		if (PIP){
```

```
                if ((vcoord_norm*INHEIGHT >= PIP_V_OFFSET)
&&
                     (vcoord_norm*INHEIGHT < PIP_V_OFFSET +
PIP_HEIGHT) &&
                     (ucoord_norm*INWIDTH >= PIP_U_OFFSET)
&&
                     (ucoord_norm*INWIDTH < PIP_U_OFFSET +
PIP_WIDTH)){

                          vcoord = (vcoord_norm*(INHEIGHT-1)
- PIP_V_OFFSET)*(INHEIGHT/PIP_HEIGHT);
                          ucoord = (ucoord_norm*(INWIDTH) -
PIP_U_OFFSET)*(INWIDTH/PIP_WIDTH) + INWIDTH;
                }
                else{
                     ucoord = ucoord_norm*(INWIDTH-2);
                     vcoord = vcoord_norm*(INHEIGHT-1);
                }
           }
           else{

                //Channel 1 on top, channel 2 on bottom
                if(OCHANNELS == 4)
                {
                     if(vf == 0) {
                          ucoord = ucoord_norm*(INWIDTH);
                          vcoord = vcoord_norm*(INHEIGHT*2);
                     }
                     else if (vf == 1) {
                          ucoord = ucoord_norm*(INWIDTH) +
INWIDTH;
                          vcoord = vcoord_norm*(INHEIGHT*2)
- INHEIGHT;
                     }
                }
                //Channel 1 on left, channel 2 on right
                if (OCHANNELS == 3)
                {
                     if (uf == 0)
                          ucoord = ucoord_norm*(INWIDTH*2);
                     else if (uf == 1)
                          ucoord = ucoord_norm*(INWIDTH*2);

                     vcoord = vcoord_norm*(INHEIGHT);
                }
           }
      }
```

```
	//Three Input Channels
	else if (ICHANNELS == 3)
	{
		//Channel 1 spread across top, Channels 2 and 3
on bottom
		if(vf == 0){
			ucoord = ucoord_norm*(INWIDTH);
			vcoord = vcoord_norm*(INHEIGHT*2);
		}
		else if (vf == 1){
			if (uf == 0){
				ucoord = ucoord_norm*(INWIDTH*2) +
INWIDTH;
				vcoord = vcoord_norm*(INHEIGHT*2) -
INHEIGHT;
			}
			else if (uf == 1) {
				ucoord = ucoord_norm*(INWIDTH*2) -
INWIDTH;
				vcoord = vcoord_norm*(INHEIGHT*2);
			}
		}
	}

	else //ICHANNELS == 4
	{
		ucoord = ucoord_norm * (INWIDTH*2-1);
		vcoord = vcoord_norm * INHEIGHT*2;
	}
}


int get_fractional_bits(float number, bool& increment)
{
	int temp = number;  //convert to integer, drop the
fractional bits
	float fractional = number - temp;

	increment = false;

	//Assume 3 bit binary representation
	int fractional_bits = 0;
	if (fractional < (0.125))
		fractional_bits = 0;
	else if (fractional < (2*.125))// + 0.125/2))
		fractional_bits = 1;
	else if (fractional < (3*.125))// + 0.125/2))
```

```
            fractional_bits = 2;
      else if (fractional < (4*.125))// + 0.125/2))
            fractional_bits = 3;
      else if (fractional < (5*.125))// + 0.125/2))
            fractional_bits = 4;
      else if (fractional < (6*.125))// + 0.125/2))
            fractional_bits = 5;
      else if (fractional < (7*.125))// + 0.125/2))
            fractional_bits = 6;
      else if (fractional < (8*.125))// + 0.125/2))
            fractional_bits = 7;     //Some rounding error
here, weighted toward 7, but we shouldn't see this case.
      else
      {
            fractional_bits = 0;
            increment = true; //Round up to next number
            //printf("Increment");
      }

      return fractional_bits;
}



void WritePerPixelLUT(int command, int CDID1, int CDID0,
int bank1Column, int bank0Column,
                              int alpha, int fracBits)
{


      int store = 0;

      int yFrac =  0;
      int xFrac = 0;

      xFrac = fracBits&0x7;
      yFrac = (fracBits&0x38)>>3;

      /*Per-Pixel LUT Format:
      |command|Reserved|CD_ID_1|CD_ID_0|bank1Column|bank0Col
umn|Alpha|Sub Y|Sub X|
      31    30 29       28 27   27 26   26 25       20 19
14 13  6 5   3 2   0
      */
      //Cache_Adr_1:  Address of Pixel 3 within cache line
for bilinear
```

```
	//Cache_Adr_1:  Address of Pixel 1 within cache line
for bilinear
	//fprintf(perPix,"%d,%d,%d,%d,%d,%d,%d,%d\n",command,C
DID1,CDID0,bank1Column,bank0Column,alpha,xFrac,yFrac);
	store =
((command&3)<<30)|((0&3)<<28)|((CDID1&1)<<27)|((CDID0&1)<<2
6)|((bank1Column&63)<<20)

	|((bank0Column&63)<<14)|((alpha&255)<<6)|((fracBits&63
));

	short storeByte3 = (store & 0xFF000000)>>24;
	short storeByte2 = (store & 0x00FF0000)>>16;
	short storeByte1 = (store & 0x0000FF00)>>8;
	short storeByte0 = (store & 0x000000FF);


	fwrite(&storeByte0, 1, 1, f2);
	fwrite(&storeByte1, 1,1,f2);
	fwrite(&storeByte2,1,1,f2);
	fwrite(&storeByte3,1,1,f2);
}


float* Rotate(int x1, int y1, int angle)
{
	float angleRad = PI*angle/180.0;
	float temp[3];
	float *out = new float[3];
	temp[2]=1;
	temp[0]=((cos(angleRad)*x1)+(-
1*sin(angleRad)*y1))/temp[2];
	temp[1]=((sin(angleRad)*x1)+(cos(angleRad)*y1))/temp[2
];
	out[0]=temp[0];
	out[1]=temp[1];
	out[2]=temp[2];
	return(out);
}




#ifndef CACHEFUNCTIONS_H
#define CACHEFUNCTIONS_H

#define PI 3.14159
```

```
const int CACHE_WIDTH = 64;
const int CACHE_LINES = 8;


int IWIDTH = 1024;
int IHEIGHT = 768;
int OWIDTH = 1024;
int OHEIGHT = 768;
int ICHANNELS = 1;
int OCHANNELS = 1;

FILE *f1,*z2,*f2,*f3,*perPix,*preFetch;

struct Pixel
{
     long address;
     int row;
     int column;
};


struct cacheLine
{
     //The start of a cache line
     int row;
     int column;
     int cid;
     int numPixels;
     Pixel data[CACHE_WIDTH];
};

struct cacheBank
{
     cacheLine line[CACHE_LINES];
};

struct cacheStart
{
     //Each Cache buffer contains 2 banks
     cacheBank bank0;
     cacheBank bank1;
};

struct sortedCache
{
     cacheLine line[CACHE_LINES*2];
```

```
	int bank[CACHE_LINES*2];
	int entries;
};

void InitializeCache(cacheStart&);
void InitializeBank(cacheBank&);
void InitializeLine(cacheLine&);
bool CacheSearchAndPlace(Pixel, int, cacheStart&, int&,
int&, int&);
bool BankSearchAndPlace(Pixel, int, cacheBank&, int&,
int&);
bool LineSearchAndPlace(Pixel, int, cacheLine&, int&);
bool FindEmptyCacheLine(Pixel, int, cacheBank&, int&,
int&);
void SortCache(cacheStart, sortedCache&);
void SortBank(cacheBank&);
void WriteCache(sortedCache);
void WriteCacheLine(cacheLine, int, int);
void InitializeSortedCache(sortedCache&);
//void RemoveFromCache(cacheBank&, int);

void DecodePixel(Pixel&);
void EncodePixel(Pixel&);
void DecodeInputPixel(Pixel&);
long DecodeOffset(long);
int DecodeCID(long);
int DecodeAlpha(long);
int DecodeIgnore(long);

bool CheckCache(Pixel, Pixel, cacheStart&, int, int&, int&,
int&, bool&, int&, int&);
void WriteHexValue(FILE*, long);


void InitializeCache(cacheStart& buffer)
{
	InitializeBank(buffer.bank0);
	InitializeBank(buffer.bank1);
}

void InitializeBank(cacheBank& bank)
{
	for(int i = 0; i < CACHE_LINES; i++)
	{
		InitializeLine(bank.line[i]);
	}
}
```

```
void InitializeLine(cacheLine& line)
{
	line.cid = -1;
	line.column = -1;
	line.row = -1;
	line.numPixels = 0;
	for(int i = 0; i < CACHE_WIDTH; i++)
	{
		line.data[i].address = -1;
		line.data[i].column = -1;
		line.data[i].row = -1;
	}
}

bool CacheSearchAndPlace(Pixel pixel, int channelID,
cacheStart& cache, int& bank, int& line, int& column)
{
	DecodePixel(pixel);
	bool bank0Placed, bank1Placed;

	int bank0Line, bank1Line;

	bank0Placed = BankSearchAndPlace(pixel, channelID,
cache.bank0, bank0Line, column);
	if(!bank0Placed)
		bank1Placed = BankSearchAndPlace(pixel,
channelID, cache.bank1, bank1Line, column);

	if(bank0Placed)
	{
		bank = 0;
		line = bank0Line;
	}
	else if (bank1Placed)
	{
		bank = 1;
		line = bank1Line;
	}
	else
	{
		bank = 99; //Return 99 to show that the pixel was
not placed in a bank
		line = bank0Line; //This value will also be 99;l
	}

	return bank0Placed || bank1Placed;
```

```
}

bool BankSearchAndPlace(Pixel pixel, int channelID,
cacheBank& bank, int& line, int& column)
{
	DecodePixel(pixel);
	bool linePlaced = false;

	line = 0;

	while(!linePlaced && line < CACHE_LINES)
	{
		linePlaced = LineSearchAndPlace(pixel, channelID,
bank.line[line], column);
		if(!linePlaced)
			line++;
	}

	if(!linePlaced)
		line = 99;

	return linePlaced;
}

bool LineSearchAndPlace(Pixel pixel, int channelID,
cacheLine& line, int& column)
{
	int placementColumn = pixel.column - line.column;

	column = 0;

	if(line.numPixels < CACHE_WIDTH && //cache line is not
full
		line.cid == channelID && //channelIDs match
		line.row == pixel.row && //rows match
		placementColumn < CACHE_WIDTH && //Pixel is
within cache
		placementColumn >= 0) //Pixel is within cache
	{
		//cout<<"Pixel should be found"<<endl;
		while(line.data[column].column != pixel.column &&
column < line.numPixels)
		{
			column++;
		}
/*		if(column == line.numPixels)
		{
```

```
                cout<<"This Shouldn't Happen"<<endl;
            }
            else if (column >= CACHE_WIDTH)
            {
                cout<<"This also shouldn't happen"<<endl;
            }
*/          if(line.data[column].column == pixel.column)
            {
            //  cout<<"Pixel already in cache"<<endl;
                return true;
            }
            else
            {
                //Should not place a pixel in the cache
here!  Make sure that other pixels will not cause
                //the cache to miss and be refilled!
                line.data[column] = pixel;
                line.numPixels++;
            //  cout<<"Pixel added to cache"<<endl;
                return true;

            }
        }
        else
        {
//          cout<<"numPixels: "<<line.numPixels<<" line.cid:
"<<line.cid<<" channelID: "<<channelID<<
//              " line.row: "<<line.row<<" pixel.row:
"<<pixel.row<<" placementColumn: "<<placementColumn<<endl;
            return false;
        }
}


//Find a cache line with no pixels in it
bool FindEmptyCacheLine(Pixel pixel, int channelID,
cacheBank& bank, int& line, int& column)
{
        //cout<<"In FindEmptyCacheLine with pixel
:"<<pixel.column<<"  "<<pixel.row<<endl;
        line = 0;
        bool lineFound = false;
        while(!lineFound && line < CACHE_LINES)
        {
            //cout<<"Line:  "<<line<<endl;
            if(bank.line[line].numPixels == 0)
            {
```

```
                //cout<<"Empty Line Found!  Line:
"<<line<<endl;
                lineFound = true;
                bank.line[line].cid = channelID;
                bank.line[line].column = pixel.column -
pixel.column%2;
                bank.line[line].row = pixel.row;
                bank.line[line].data[0].address =
pixel.address - pixel.column%2;
                bank.line[line].data[0].column =
pixel.column - pixel.column%2;
                bank.line[line].data[0].row = pixel.row;
                bank.line[line].numPixels++;
                if(pixel.column%2 == 1)
                {
                     bank.line[line].numPixels++;
                     column++;
                     bank.line[line].data[1].address =
pixel.address;
                     bank.line[line].data[1].column =
pixel.column;
                     bank.line[line].data[1].row =
pixel.row;
                }
                //cout<<"Found Empty Cache Line"<<endl;
           }
           else
           {
                line++;
                //cout<<"Empty Line not found.  Next line to
check:  "<<line<<endl;
           }
     }

     return lineFound;
}

void SortCache(cacheStart cache, sortedCache& sorted)
{
     SortBank(cache.bank0);
     SortBank(cache.bank1);

     int bank0Index = 0;
     int bank1Index = 0;

//   sorted.line[0] = cache.bank0.line[0];
//   sorted.line[1] = cache.bank1.line[0];
```

```
     if(abs(cache.bank1.line[0].row -
cache.bank0.line[0].row) > 1)

     printf("%d,%d\n",cache.bank0.line[0].row,cache.bank1.l
ine[0].row);
     //Merge the two banks together
     for(int i = 0; i < CACHE_LINES*2; i++)
     {
          if(bank0Index < CACHE_LINES)
          {
               if(bank1Index < CACHE_LINES)
               {
                    //Compare the line row numbers
                    if(cache.bank0.line[bank0Index].row <=
cache.bank1.line[bank1Index].row)
                    {
                         sorted.line[i] =
cache.bank0.line[bank0Index];
                         sorted.bank[i] = 0;
                         bank0Index++;
                    }
                    else
                    {
                         sorted.line[i] =
cache.bank1.line[bank1Index];
                         sorted.bank[i] = 1;
                         bank1Index++;
                    }

               }
               else //bank1 is done being processed
               {
                    sorted.line[i] =
cache.bank0.line[bank0Index];
                    sorted.bank[i] = 0;
                    bank0Index++;
               }
          }
          else if (bank1Index < CACHE_LINES) //Bank 0 is
done being processed
          {
               sorted.line[i] =
cache.bank1.line[bank1Index];
               sorted.bank[i] = 1;
               bank1Index++;
          }
```

```
      }//End of For loop

}


void SortBank(cacheBank & bank)
{
      for(int i = 0; i < CACHE_LINES -1; i++)
      {
            int min = i;
            for (int j = i+1; j <CACHE_LINES; j++)
            {
                  if(bank.line[j].row < bank.line[min].row)
                  {
                        min = j;
                  }
            }
            if (i != min)
            {
                  cacheLine swapLine = bank.line[i];
                  bank.line[i] = bank.line[min];
                  bank.line[min] = swapLine;
            }
      }
}

void WriteCache(sortedCache cache)
{
      int bank0LineNum = 0;
      int bank1LineNum = 0;
      for(int i = 0; i < CACHE_LINES*2; i++)
      {
            if(cache.bank[i] == 0)
            {
                  WriteCacheLine(cache.line[i], cache.bank[i],
bank0LineNum);
                  bank0LineNum++;
            }
            else if (cache.bank[i] == 1)
            {
                  WriteCacheLine(cache.line[i], cache.bank[i],
bank1LineNum);
                  bank1LineNum++;
            }
      }
//    if(abs(cache.line[1].row-cache.line[0].row)>1)
```

```
//
	printf("%d,%d\n",cache.line[0].row,cache.line[1].row);
}

void WriteCacheLine(cacheLine line, int bank, int lineNum)
{
	unsigned int store = 0;
	Pixel cacheMarker;


	if(line.numPixels%2 == 1)
		line.numPixels++;

	while(line.numPixels > CACHE_WIDTH)
		line.numPixels -= 2;

	//SET NUMBER OF PIXELS TO CACHE WIDTH FOR THESIS DATA
	//line.numPixels = CACHE_WIDTH;

	line.numPixels = line.numPixels >> 1;

	line.numPixels--;

	//Possible bug in Ben's code, make sure that
line.numPixels >= 4
	if(line.numPixels < 4)
		line.numPixels = 4;

	//Make sure a -1 doesn't sneak in
	if(line.column < 0)
		line.column = 0;
	if(line.row < 0)
		line.row = 0;
	if(line.cid < 0)
		line.cid = 0;

	if(line.column > IWIDTH)
		printf("Column too high! %d
%d\n",line.column,line.row);
		//cout<<"Column too high! "<<line.column<<"
"<<line.row<<endl;
	if(line.row > IHEIGHT)
		printf("Row too high! %d
%d\n",line.column,line.row);
		//cout<<"Row too hight! "<<line.column<<"
"<<line.row<<endl;
```

```
	cacheMarker.column = line.column;
	cacheMarker.row = line.row;
	EncodePixel(cacheMarker);

	//fprintf(preFetch,"%d,%d,%d,%d,%d,%d\n",line.numPixel
s,line.cid,line.column, line.row,bank, lineNum);

	//Prefetch LUT Format
	// |Reserved|Pixels to Load| CID
|Offset|CD_BANK|CD_ID|
	// |31    31|30               26|25 24|23   4|3
3|2   0|

	store =
((line.numPixels&0x1F)<<26)|((line.cid&0x3)<<24)|(((cacheMa
rker.address&0x1FFFFE)>>1)<<4)|((lineNum&1)<<3)|(bank&0x7);

	//fwrite(&store, sizeof(long), 1, f3);
	//fwrite(&store, 2, 2, f3);
	//short storeLower = store &0x0000FFFF;
	short storeByte3 = (store & 0xFF000000)>>24;
	short storeByte2 = (store & 0x00FF0000)>>16;
	short storeByte1 = (store & 0x0000FF00)>>8;
	short storeByte0 = (store & 0x000000FF);


	fwrite(&storeByte0, 1, 1, f3);
	fwrite(&storeByte1, 1,1,f3);
	fwrite(&storeByte2,1,1,f3);
	fwrite(&storeByte3,1,1,f3);
	//fwrite(&store, 2, 1, f3);

	////////////////////////////////////////////
	//Function Supports Ben's FPGA Testing
	////////////////////////////////////////////
	//WriteHexValue(f3, store);

}


void InitializeSortedCache(sortedCache& cache)
{
	for(int i = 0; i < CACHE_LINES*2; i++)
	{
		cache.bank[i] = 0;
		cache.line[i].cid = 0;
		cache.line[i].column = 0;
```

```
            cache.line[i].numPixels = 0;
            cache.line[i].row = 0;
      }
      cache.entries = 0;
}

//Get the address column and row from the raw address
void DecodePixel(Pixel &pix)
{
      pix.column = pix.address%OWIDTH;
      pix.row = (pix.address - pix.column)/OWIDTH;
}

//Get the raw address of the pixel from the row and column
void EncodePixel(Pixel &pix)
{
      pix.address = pix.row*OWIDTH + pix.column;
}


bool CheckCache(Pixel currentPixel, Pixel ptrPixel,
cacheStart& cache, int channelID, int& bank0Column,
                     int& bank1Column, int& command, bool&
nearestNeighbor, int& bank0Line, int& bank1Line)
{
      Pixel x2, y1, y2;

      //Make sure we have row/column information
      DecodeInputPixel(currentPixel);

      bool currentPlaced, x2Placed, y1Placed, y2Placed,
cacheHit;

      bool cacheFull = false;

      bool nextHit;

      int currentLine, x2Line, y1Line, y2Line;
      int currentBank, x2Bank, y1Bank, y2Bank;
      int currentColumn, x2Column, y1Column, y2Column;

      //Find other 3 pixels for bilinear.  Normally assume
down and right in the array, but
      //on edges compensate by going the other way to avoid
accessing a ptr value that does not exist
      if((ptrPixel.row < OHEIGHT - 1) && (ptrPixel.column <
OWIDTH -1))
```

```
     {
               //If input boundary, assume nearest neighbor
          if((currentPixel.column == IWIDTH -
1)||(currentPixel.row == IHEIGHT -1))
          {
               x2.address = currentPixel.address;
               DecodeInputPixel(x2);
               y1.address = currentPixel.address;
               DecodeInputPixel(y1);
               y2.address = currentPixel.address;
               DecodeInputPixel(y2);
               nearestNeighbor = true;
          }
          else {
               x2.address = currentPixel.address+1;
               DecodeInputPixel(x2);
               y1.address = currentPixel.address+OWIDTH;
//OWIDTH?  IWIDTH?
               DecodeInputPixel(y1);
               y2.address = currentPixel.address+OWIDTH+1;
               DecodeInputPixel(y2);
               nearestNeighbor = false;
          }
     }
     else //if on the boundary, assume nearest neighbor
interpolation
     {
                         //If input boundary, assume
nearest neighbor
          if((currentPixel.column == IWIDTH -
1)||(currentPixel.row == IHEIGHT -1))
          {
               x2.address = currentPixel.address;
               DecodeInputPixel(x2);
               y1.address = currentPixel.address;
               DecodeInputPixel(y1);
               y2.address = currentPixel.address;
               DecodeInputPixel(y2);
               nearestNeighbor = true;
          }
          else {
               x2.address = currentPixel.address;
               DecodeInputPixel(x2);
               y1.address = currentPixel.address;
               DecodeInputPixel(y1);
               y2.address = currentPixel.address;
               DecodeInputPixel(y2);
```

```
                    nearestNeighbor = true;
               }
          }


          //Check if currentPixel and x2, y1, and y2 are in the
cache
          //Assume:  column0 == column1
//        if (!nearestNeighbor)
//        {
               currentPlaced = BankSearchAndPlace(currentPixel,
channelID, cache.bank0, currentLine, currentColumn);
               x2Placed = BankSearchAndPlace(x2, channelID,
cache.bank0, x2Line, x2Column);

               if(!nearestNeighbor) {
                    y1Placed = BankSearchAndPlace(y1, channelID,
cache.bank1, y1Line, y1Column);
                    y2Placed = BankSearchAndPlace(y2, channelID,
cache.bank1, y2Line, y2Column);
               }
               else{
                    y1Placed = BankSearchAndPlace(y1, channelID,
cache.bank0, y1Line, y1Column);
                    y2Placed = BankSearchAndPlace(y2, channelID,
cache.bank0, y2Line, y2Column);
               }
               //Don't need the following code with only 1 cache
line!

               if(!currentPlaced)
               {
                    //cout<<"currentPixel Not Placed:
"<<ptrPixel.column<<"  "<<ptrPixel.row<<endl;
                    cacheFull =
!FindEmptyCacheLine(currentPixel, channelID, cache.bank0,
currentLine, currentColumn);
                    if(!cacheFull)//Should be able to place x2
now if cacheLine is not full
                         cacheFull = !BankSearchAndPlace(x2,
channelID, cache.bank0, x2Line, x2Column);
               }

               if(!y1Placed)
               {
```

```
                cacheFull = cacheFull |
!FindEmptyCacheLine(y1, channelID, cache.bank1, y1Line,
y1Column);
                if(!cacheFull)//Should be able to place y2
now if cacheLine is not full
                    cacheFull = !BankSearchAndPlace(y2,
channelID, cache.bank1, y2Line, y2Column);
          }


          cacheHit = (currentPlaced && x2Placed && y1Placed
&& y2Placed)||!cacheFull;

     if(cacheHit)
     {
          //Put the pixels in the cache!


          //Find the relative column that the pixel is in
          //column = currentPixel.column - cache.column0;
          bank0Column = currentColumn;
          bank1Column = y1Column;

          command = 0;
          /*//Determine which command to use
          if(nextHit)
               command = 0;
          else
               command = 1;
               */

          //Last pixel should have a command of 1
          if(ptrPixel.column == OWIDTH-1 && ptrPixel.row ==
OHEIGHT -1)
          {
               command = 1;
               bank0Column = currentColumn;
               bank1Column = y1Column;
               //column = currentPixel.column -
cache.column0;
               cacheHit = true;
          }
          bank0Line = currentLine;
          bank1Line = y1Line;
     }
     else
     {
```

```
          //Last pixel should have a command of 1
          if(ptrPixel.column == OWIDTH-1 && ptrPixel.row ==
OHEIGHT -1)
          {
               command = 1;
               bank0Column = currentColumn;
               bank1Column = y1Column;
               bank1Line = 0;
               bank0Line = 0;
               //column = currentPixel.column -
cache.column0;
          }
          else
          {
               command = 1;  //Switch to 0?
               bank0Column = currentPixel.column%2;
               bank1Column = y1.column%2;
               bank1Line = 0;  //Set to 0 for now at least
               bank0Line = 0;
               //column = currentPixel.column%2;
//             if(abs(currentPixel.address-y1.address) >
OWIDTH)
                    //printf("%d,%d\n",currentPixel.row,
y1.row);
          }
     }


     return cacheHit;
}


//Get the address column and row from the raw address
void DecodeInputPixel(Pixel &pix)
{
     pix.column = pix.address%IWIDTH;
     pix.row = (pix.address - pix.column)/IWIDTH;
}


long DecodeOffset(long store)
{
     return store&0x001FFFFF;
}

int DecodeCID(long store)
```

```
{
	return (store>>21)&0x3;
}

int DecodeAlpha(long store)
{
	return(store>>23)&0xFF;
}

int DecodeIgnore(long store)
{
	if((store&0xFF000000)>>24 == 0x80)
		return 1;
	else
		return 0;
}


void WriteHexValue(FILE* f, long store)
{
	unsigned int a1,a2,a3,a4;
	unsigned char* p;
	unsigned int check=0;
//	unsigned int value =store;

	char lowByte[2];
	char highByte[2];
	char lowByte2[2];
	char highByte2[2];

	a1=(store&0xff000000)>>24;
	a2=(store&0x00ff0000)>>16;
	a3=(store&0x0000ff00)>>8;
	a4=(store&0x000000ff)>>0;

	itoa(a4, lowByte, 16);
	itoa(a3, highByte, 16);

/*	if(lowByte[1] == NULL)
		lowByte[1] = '0';
	if(highByte[1] == NULL)
		highByte[1] = '0';
*/
	if(a4 == 0)
	{
		lowByte[0] = '0';
		lowByte[1] = '0';
```

```
        }
        else if (a4 <= 0xf)
        {
             lowByte[1] = lowByte[0];
             lowByte[0] = '0';
        }
        if(a3 == 0)
        {
             highByte[0] ='0';
             highByte[1] = '0';
        }
        else if (a3 <= 0xf)
        {
             highByte[1] = highByte[0];
             highByte[0] = '0';
        }

        char lowNib[] = {highByte[0], highByte[1], lowByte[0],
lowByte[1], '\n'};// lowByte[1], lowByte[0],
highByte[1],highByte[0], '\n'};

        itoa(a2, lowByte2, 16);
        itoa(a1, highByte2, 16);

        if(a2 == 0)
        {
             lowByte2[0] = '0';
             lowByte2[1] = '0';
        }
        else if (a2 <= 0xf)
        {
             lowByte2[1] = lowByte2[0];
             lowByte2[0] = '0';
        }
        if (a1 == 0)
        {
             highByte2[0] = '0';
             highByte2[1] = '0';
        }
        else if (a1 <= 0xf)
        {
             highByte2[1] = highByte2[0];
             highByte2[0] = '0';
        }
```

```
	char highNib[] = {highByte2[0], highByte2[1],
lowByte2[0], lowByte2[1], '\n'};// lowByte2[1],
lowByte2[0], highByte2[1], highByte2[0], '\n'};

	//unsigned char nl = '\n';

//	p = &a4;
	fwrite(lowNib, sizeof(unsigned
char),sizeof(lowNib),f);
	fwrite(highNib, sizeof(unsigned
char),sizeof(highNib),f);

}


#endif
```

Appendix B: Cache Simulation Code

The following code, written in C++, is used to simulate the behavior of the Pixel Router's SDRAM, cache, and FGPA. The code processes a Look Up Table and determines the corresponding output image frame rate.

```
#include<string.h>
#include<stdio.h>
#include<iostream>
#include<fstream>
#include<sstream>
#include<stdlib.h>
#include<queue>
#include<math.h>
//#include "cache.h"

using namespace std;

struct Pixel
{
	long address;
	int row;
	int column;
};

struct PerPixelEntry
```

```
{
	int command;
	int CD_ID1;
	int CD_ID0;
	int addr1;
	int addr0;
	int alpha;
	int suby;
	int subx;

};



void ReadFiles(); //Returns number of blank pixels
int ProcessPixels(void);
//void DecodePixel(Pixel&);
bool RefillEntry(unsigned int);
void ReadPrefetchEntry(int, int&, int&, int&);
void DecodeOffset(unsigned int, int&, int&);
int DecodePixels(unsigned int);
bool DecodePerPixelEntry(unsigned int, PerPixelEntry&);
bool ReadPerPixelEntry(int, PerPixelEntry&);
float AverageRotation(void);
float StandardDevRotation(float);

const int PI = 3.14159265;

//LUT length = Output Width * Output Height + 1
const int OWIDTH = 1024;
const int OHEIGHT = 768;

const int IWIDTH = 1024;
const int IHEIGHT = 768;

int CACHE_WIDTH = 64;
int CACHE_LINES = 2;

//Clock Frequency
const float CLK = 133000000.0;

const int T_OPEN_ROW = 12; //Number of cycles to close
current DRAM row and open a new one
const float T_READ_PIXEL = 0.5; //Number of clock cycles to
read a pixel on a cache line
```

```
unsigned int ptr [OWIDTH*OHEIGHT]; //Stores data read from
per-pixel LUT
unsigned int cache_ptr[OWIDTH*OHEIGHT]; //Prefetch LUT
entries
float rotation_amount[OWIDTH*OHEIGHT];

int prefetchSize = 0;
int cacheMisses = 0;
int cacheHits = 0;

struct address
{
     int col;
     int row;
};

int main (int argc, char * argv[])
{
     //int cache_width[4] = {8, 16, 32, 64};
     //int cache_lines[4] = {2, 4, 6, 8};

     int numCycles;

     string outFile = "Cache Simulation Results P2L.csv";
     ofstream ofile(outFile.c_str(), ios::out);
     float frameRate = 0;

     /*if (argc != 2)
     {
          cout<<"Correct usage:  Bilinear Cache
Simulation.exe [perPixelLUT.txt]"<<endl;
          exit(1);
     }

     string PerPixelLUT = argv[1];
*/
     ofile<<"Cache Width, Cache Lines, # Pixels, Clock
Cycles, Frame Rate"<<endl;

     //Get the data from the Prefetching LUT and store it
in ptr
     ReadFiles();

     numCycles = ProcessPixels();

     frameRate = 1/(4*numCycles*1/CLK);
     //TODO:  Include Hit-rate
```

```
	ofile<<CACHE_WIDTH<<","<<CACHE_LINES<<","<<OWIDTH*OHEI
GHT<<","<<numCycles<<","<<

	1/(4*numCycles*1/CLK)<<","<<(cacheHits*100.0)/(cacheHi
ts+cacheMisses*1.0)<<","<<
		(1024*768*4*2*frameRate)/(CLK*2)<<"\n";

	ofile<<"\n\n";

	ofile.close();

	float rotationAverage = AverageRotation();
	float standardDeviation =
StandardDevRotation(rotationAverage);

	cout<<"Average Rotation: "<<rotationAverage<<endl;
	cout<<"Standard Deviation: "<<standardDeviation<<endl;

	cout<<"Finished!"<<endl;
	int var;
	cin>>var;
	return 0;
}

void ReadFiles()
{
	FILE * perPixelLUT = fopen("lut1.txt", "rb");
	FILE * prefetchLUT = fopen("cache_lut1.txt", "rb");

	if(perPixelLUT == 0) //Can't open the file
	{
		cout<<"File not located"<<endl;
		exit(2);
	}

	int fileNum = 0;
	int outWidth = 0;
	int outHeight = 0;
	int dummy = 0;
	int extraPrefetchEntries = 0;
	int extraPerPixelEntries = 0;

	//int prefetchSize= 0;

	//Read in per-pixel header information
	fread(&fileNum, sizeof(int), 1, perPixelLUT);
	fread(&outWidth, sizeof(int), 1, perPixelLUT);
```

```
	fread(&outHeight, sizeof(int), 1, perPixelLUT);
	fread(&dummy, sizeof(int), 1, perPixelLUT);

	//Read in the data from the LUT
	for(int i = 0; i < OWIDTH*OHEIGHT; i++)
		fread(&ptr[i], sizeof(int), 1, perPixelLUT);
	try {
		cout<<"Per Pixel Extra Entries"<<endl;
		for (int j = 0; j < 100; j++)
		{
			fread(&dummy, sizeof(int), 1, perPixelLUT);
			cout<<dummy<<endl;
		}
	}
	catch (char* str)
	{
		cout<<"No extra data in per pixel LUT"<<endl;
	}

	//Read in prefetching header information
	fread(&dummy, sizeof(int), 1, prefetchLUT);
	fread(&prefetchSize, sizeof(int), 1, prefetchLUT);
	fread(&dummy, sizeof(int), 1, prefetchLUT);

	for(int j = 0; j < prefetchSize/4; j++)
		fread(&cache_ptr[j], sizeof(int), 1,
prefetchLUT);
	try {
		cout<<"Prefetch Extra Entries"<<endl;
		for (int k = 0; k < 100; k++) {
			fread(&dummy, sizeof(int), 1, prefetchLUT);
			cout<<dummy<<endl;
		}
	}
	catch (char* str)
	{
		cout<<"No extra data in prefetch LUT"<<endl;
	}
	fclose(perPixelLUT);
	fclose(prefetchLUT);
}

//Return the number of clock cycles required for processing
the LUT
int ProcessPixels()
{
```

```
        int pixels1 = 0;
        int pixels2 = 0;
        int pixelsLast = 0;
        int refillCycles = 0;
        int pixelsSinceRefill = 0;
        int numCycles = 0;

        int prefetchIndex = 0;
        int lastRowOpened = 0;
        int prefetchRow = 0;
        int prefetchCol = 0;
        int lastPrefetchRow = 0;

        int perPixelCommand = 0;
        int perPixelIndex = 0;
        int refreshCommands = 0;
        int prefetchEntries = 0;
        int pixelsSinceRefresh = 0;

        int additionalPerPixelEntries = 0;

        int rowAdvantage = 0;
        int ignorePixels = 0;

        int pixelRotation = 0;
        int lastLoadedRow = 0;
        int firstRow = 0;
        int rotationPixels = 0;

        int PixelsToLoad = 0;

        PerPixelEntry entry;

        for(int i = 0; i < prefetchSize/4; i++)
        {

            ReadPrefetchEntry(i, prefetchRow, prefetchCol,
pixels1);
            prefetchEntries++;
            firstRow = prefetchRow;

            PixelsToLoad = pixels1;

            //Used to simulate no PixelsToLoad
            //pixels1 = CACHE_WIDTH;
```

```
          if(prefetchRow == lastRowOpened)
          {
               refillCycles = (pixels1)*T_READ_PIXEL;
//numRefillCyclesOneRow;
               rowAdvantage++;
          }
          else
               refillCycles = T_OPEN_ROW +
(pixels1)*T_READ_PIXEL; //numRefillCycles;

          if (prefetchRow > OHEIGHT)
               cout<<"Prefetch Row Too Big! (first row)
"<<prefetchRow<<"  "<<prefetchCol<<endl;

          lastRowOpened = prefetchRow;

          //read the second row
          i++;
          ReadPrefetchEntry(i, prefetchRow, prefetchCol,
pixels2);

          //if (prefetchRow > OHEIGHT)
               //cout<<"Prefetch Row Too Big! (second
row)"<<endl;

          if(prefetchRow == lastRowOpened)
          {
               refillCycles += (pixels1)*T_READ_PIXEL;
//numRefillCyclesOneRow;
               rowAdvantage++;
               //cout<<"Shouldn't happen  "<<prefetchRow<<"
"<<prefetchCol<<endl;
          }
          else
               refillCycles += T_OPEN_ROW +
(pixels1)*T_READ_PIXEL; //numRefillCycles;

          lastRowOpened = prefetchRow;


          pixelsSinceRefresh = 0;


          numCycles += refillCycles;
     }

     for (int i = perPixelIndex; i < OHEIGHT*OWIDTH; i++)
```

```
        {
            additionalPerPixelEntries ++;
        }

        return numCycles;
}

bool RefillEntry(unsigned int lutEntry)
{
        int refill;

        //cout<<lutEntry<<endl;

        refill = (lutEntry&0x40000000)>>30;

        if (refill == 1)
            return true;
        else
            return false;
}

void ReadPrefetchEntry(int index, int& row, int& col, int&
pixels)
{
        DecodeOffset(cache_ptr[index], row, col);
        pixels = DecodePixels(cache_ptr[index]);
}

void DecodeOffset(unsigned int prefetchEntry, int& row,
int& col)
{
        int offset = ((prefetchEntry&0x000FFFFF0)>>4)*2;
        col = offset%IWIDTH;
        row = (offset-col)/IWIDTH;
}

int DecodePixels(unsigned int prefetchEntry)
{
        int pixelsLoaded =
(((prefetchEntry&0x7C000000)>>26)+1)*2;
        return pixelsLoaded;
}

bool ReadPerPixelEntry(int index, PerPixelEntry & entry)
{
        return DecodePerPixelEntry(ptr[index], entry);
}
```

```
bool DecodePerPixelEntry(unsigned int perPixelEntry,
PerPixelEntry& entry)
{
	entry.command = ((perPixelEntry&0xC0000000)>>30);
	entry.CD_ID1 = ((perPixelEntry&0x08000000)>>27);
	entry.CD_ID0 = ((perPixelEntry&0x04000000)>>26);
	entry.addr1 = ((perPixelEntry&0x03F00000)>>20);
	entry.addr0 = ((perPixelEntry&0x000FC000)>>14);
	entry.alpha = ((perPixelEntry&0x00003FC0)>>6);
	entry.suby = ((perPixelEntry&0x00000038)>>3);
	entry.subx = ((perPixelEntry&0x00000007));

	if (entry.CD_ID0 != 0 || entry.CD_ID1 != 0 ||
(entry.command != 0 && entry.command != 1))
	{
		//something is wrong with the entry, return false
		return false;
	}
	else
		return true;
}


float AverageRotation()
{
	float rotationSum = 0;
	for (int i = 0; i < OWIDTH*OHEIGHT; i++) {
		rotationSum += rotation_amount[i];
	}
	return rotationSum/(OWIDTH*OHEIGHT);
}

float StandardDevRotation(float avg)
{
	float standardDevSum = 0.0;
	for (int i = 0; i < OWIDTH*OHEIGHT; i++) {
		standardDevSum +=
(float)pow((float)(rotation_amount[i]-avg), 2);
	}
	return sqrt(standardDevSum/(OWIDTH*OHEIGHT));
}
```

REFERENCES

[1] Michael Brown, Aditi Majumder, Ruigang Yang, "Camera-Based Calibration Techniques for Seamless Multi-Projector Displays", IEEE Transactions on Visualization and Computer Graphics, Vol. 11, No. 2, March/April 2005, pp. 193-206.

[2] Ruigang Yang, Anselmo Lastra, "Anywhere Pixel Compositor", 34th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Article No. 10, 2007.

[3] Vijai Raghunathan, "An Effective Cache for the Anywhere Pixel Router", University of Kentucky 2007.

[4] Ben Klass, Dave Matthews, "Pixel Router FPGA Design Specification", Verien Design Group, LLC, 27 June, 2010. (Linked, available here)

[5] Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing, Second Edition", Upper Saddle River, NJ: Prentice Hall, 2002, pp 272-275.

[6] Micron Technology, Inc. "Double Data Rate (DDR) SDRAM" MT46V32M16 datasheet, 2009. (Available at: http://download.micron.com/pdf/datasheets/dram/ddr/512MBDDRx4x8x16.pdf)

[7] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach, Fourth Edition", San Francisco, CA: Morgan Kaufman, 2007, pp 293-309.

[8] Synergy Global Technology Inc., PS2002/TC-1U35 datasheet (Available at: http://www.rackmountmart.com/dataSheet/ps2002.pdf)

[9] Federal Communications Commission, Title 47 Code of Federal Regulations, Part 15, Subpart B, Sections 107-109, Washington, DC, 2009.

[10] Intertek Testing Services NA, Inc., "Pixel router test report", Lexington, KY, Report Number 100158311LEX-001, 28 July 2010. (Linked, available here)

[11] Tyco Electronics, "EF Series Compact RFI Filter with IEC Connectors", 15EF1F datasheet. (Available at: http://datasheet.octopart.com/15EF1F-Tyco-Electronics-datasheet-35071.pdf)

[12] Office of the Under Secretary of Defense (Comptroller), Department of Defense Fiscal Year 2011 Budget Procurement Programs, 2010. (Available at: http://comptroller.defense.gov/defbudget/fy2011/fy2011_p1.pdf)

[13] Office of the Under Secretary of Defense (Comptroller), Department of Defense Fiscal Year 2011 Budget RDT&E Programs, 2010. (Available at: http://comptroller.defense.gov/defbudget/fy2011/fy2011_r1.pdf)

[14] Geoffroy Chateauneuf, Proposal for Fabrication and Assembly of: DVI Router 4, Creative Exchange, Inc., 24 July 2008. (Linked, available here)

VITA

Steven Dominick was born in Pensacola, Florida on August 17, 1984. He graduated magna cum laude from the University of Kentucky with a bachelor's degree in Electrical Engineering in 2007 while on full scholarship for his entire course of study. He has completed engineering internships at ADTRAN, Inc. and Lexmark International Inc., and has worked as an engineer for the University of Kentucky Center for Visualization and Virtual Environments. While at ADTRAN, Inc., he was named the Co-op Ambassador to the University of Kentucky. He demonstrated the Pixel Router the 5th ACM/IEEE International Workshop on Projector Camera Systems 2008, and published a corresponding document in the conference proceedings.