Project Proposal
Augmented Reality Image Processing System

Logan P. Williams & José E. Cruz Serrallés

November 03, 2011

1 Abstract

In our project, we will implement an augmented reality system that can overlay a digital image on video of a real world environment. We begin by reading NTSC video from a video camera and storing it in ZBT SRAM. A picture frame with colored markers on the corners is held in front of the camera. We then perform chroma-based object recognition to locate the co-ordinates of the corners. Using these co-ordinates, we apply appropriate translation, scaling, rotation, and anti-aliasing FIR filters to fit the image to the boundary of the frame. If time allows, we will use a non-linear projective transformation to correct for perspective. We then output VGA video of the original captured image, with the processed image overlayed on top of the frame. The overlayed image (the "augmentation") can be arbitrary. When this image is the frame of video that was previously displayed, we call the system "recursive", as we obtain the same image contained within itself.

2 Top-Level Block Diagram

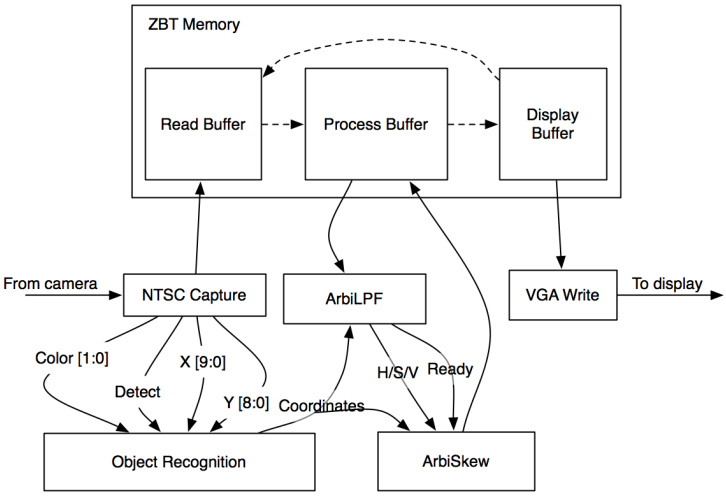

Figure 1: The block diagram of the augmented reality system

3 Submodules

3.1 NTSC Capture (Logan)

The NTSC capture module is almost unmodified from the code provided by 6.111. It takes as input an NTSC video signal, and writes pixels into ZBT Memory.

There are two modifications. The first allows the module to capture and store color data, converting it from Y/Cr/Cb to H/S/V. The second modification is added to support object recognition. When the Capture module sees a pixel of a hue that matches the target (blue, green, red, and yellow), it sends information (the color and its X/Y location) to the Object Recognition module.

3.2 ZBT Memory (José)

The ZBT Memory module stores three 640x480 images in ZBT RAM. ZBT RAM was chosen instead of BRAM because the data in three 640x480 RGB images vastly exceeds our BRAM capacity. These three images will be (1) the image that is currently being captured, (2) the image that was just captured and to which skew will write its pixels, and (3) the image that is currently displayed. These three images will be henceforth referred to as "capturing image", "processing image", and "displaying image", respectively. The inputs to ZBT Memory are (1) the next pixel to be written to capturing image, (2) the next pixel to be written to processing image and (3) its index, (4) the index of the next pixel to be read from processing image, (5) the index of the next pixel to be read from displaying image as requested by the VGA Write module, and (6) the set of indices of the next set of pixels to be read from displaying image as requested by the ArbiLPF module. The outputs of ZBT Memory are (1) a pulse indicating whether a pixel was written

to capturing image, (2) a pulse indicating whether a pixel was written to processing image, (3) the current pixel to be read from processing image, (4) a pulse indicating that this output was updated, (5) the current pixel to be read from displaying image as requested by the VGA Write module, (6) a pulse indicating that this output was updated, (7) the current set of pixels to be read from displaying image as requested by the ArbiLPF module, and (8) a pulse indicating that this output was updated.

The three images will be split between two ZBT RAM modules; as such, the ZBT Memory module will keep track of which image is in which block and intelligently return the correct pixel for each respective image given a relative index. Every NTSC capture refresh cycle (ie, every 1/30 seconds), the ZBT Memory module will switch the current processing image to be the new displaying image, the current displaying image to be the new capturing image, and the current capturing image to be the new processing image. Thus, reads and writes will be completely mandated by submodules and no data will have to be shifted around, improving efficiency.

The single write or read per cycle constraint that is imposed by each ZBT RAM module necessitates the use of a much higher clock speed than the standard 21.175MHz used for 640x480 at 60Hz output. Therefore, the clock speed will have to be at least approximately 120MHz for appropriate latency for this module. The ArbiLPF module, the ArbiSkew module, and the Object Recognition module should be run on the order of 90MHz, because this clock frequency allows for multiplications to be executed safely and allows the computationally intensive submodules enough time to complete all of their calculations.

The ZBT Memory will be a challenge to test thoroughly. The alternation of image locations will be tested with one testbench, and small sample images. Another testbench will be written to test the interface between the two RAM blocks and all of the read and write requests that they will be receive from the other submodules. Extensive care will be taken to ensure fairness among all of the submodules when contesting the memory, ie, not module should be favored too heavily among the other submodules. With a high enough clock frequency, this contention problem should not be a big issue.

3.3 Object Recognition (Logan)

The Object Recognition module collects "interesting" pixels located by the NTSC Capture module, and calculates the center of mass of each color, to find the location of the corners of the picture frame. It takes as input (1) the color of a detected pixel, (2) a flag that goes high for one clock cycle when a pixel is detected, (3) the X/Y coordinates of the pixel, and (4) a flag that goes high when the entire frame has been captured. It produces as output four sets of X/Y coordinates, one of the center of mass of each color.

The center of mass will be calculating with a simple linear weighting scheme, averaging the X and the Y coordinate for each pixel independently to find the center X/Y location, which are used by the ArbiLPF and the ArbiSkew module.

3.4 ArbiLPF (José)

The inputs to ArbiLPF are (1) the downsampling coefficient, (2) the index of the pixel in displayed image to be filtered given this downsampling factor, (3) the set of pixels around the filtered pixels required for filtering, and (4) the pulse from the memory module. ArbiLPF applies a two-dimensional low-pass filter to this pixel by using surrounding pixels to calculate the convolution sum. The radial cutoff frequency of this

2D filter is of $\frac{\pi}{M}$, in order to avoid aliasing in the ArbiSkew module. The outputs of ArbiLPF are (1) the pixel values of the output of the lowpassed version of this image, sampled at the given index, and (2) the indices of the set of pixels that are needed for filtering.

Based on the downsampling factor M, the filter will select a set of coefficients from a lookup table and convolve the image values with these coefficients. This table of coefficients will correspond to the coefficients of 2D extrapolations of 1D FIR Parks-McClellan filters with cutoff frequencies of $\frac{\pi}{M}$. Due to the limited number of multipliers on the FPGA and the single-input, single-output of the RAM module, these 2D filters will be constrained to have at most 16 coefficients, which constrains the one-dimensional filters to have at most 4 coefficients. Due to these constraints, the ripple and transition width specifications of the 1D filters will have to be lax. The radial symmetry of these 2D filters will be exploited to reduce the number of required multiplications by a factor of 4, to at most 4 multiplications per color per pixel or 12 multiplications per pixel.

Due to the single-input, single-output nature of the RAM, the clock frequency of this module will have to be greater than 80MHz. Given relatively little contention from other blocks, ArbiLPF will elapse at least 9 cycles at 90MHz per pixel, yielding a latency of 0.03072 seconds or a little less than one NTSC refresh period. ArbiLPF will also have to perform roughly eight additions per cycle, but the timing constraints imposed by these additions are negligible when compared to the multiplications.

ArbiLPF will be tested by crafting a testbench module that accepts an arbitrary 640x480 image and outputs the output of the filter. Initially, an image with an impulse at the center will be used, which ideally should cause the filter to output the filter coefficients that are used. As basic functionality is tested, more complicated images will be used. Eventually, complex images will be processed both with the testbench and with MATLAB and will be compared using the 2D Fourier plots of these two outputs.

3.5 ArbiSkew (Logan)

The inputs to ArbiSkew are (1) the hue, saturation, and value of the last pixel produced by ArbiLPF, (2) a ready signal held high for one clock signal when ArbiLPF has processed a new pixel, and (3) the four coordinates of the corners of the frame provided by the Object Recognition module.

This function maps the original rectangular image to any convex quadrilateral, provided that all sides of the destination quadrilateral are shorter than the original, which is inherent in the overall system. A graphic representation of the transformation is shown in the figure below:

Mathematically, the algorithm works as follows:

1. Calculate the distance of line $\overline{A\prime D\prime}$ and assign it to d_{ad}.
2. Do the same for $\overline{B\prime C\prime}$ and assign it to d_{bc}.
3. Create two "iterator points," point I_A and I_B initially located at $A\prime$ and $B\prime$.
4. Let $o_x = 0$ and $o_y = 0$
5. Calculate the distance between the iterator points, assign it to d_i.
6. Create a third iterator point, I_C at the location I_A.
7. Assign the pixel value of I_C to pixel (o_x, o_y) in the original image.
8. Move I_C along line $\overline{I_A I_B}$ by an amount $= \frac{d_i}{width_{original}}$.
9. Increment o_x.

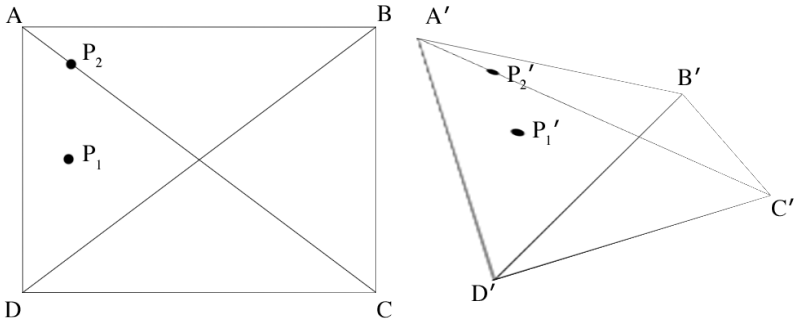

Figure 2: A visual representation of the result of the ArbiSkew module. Input is on the left, a possible output, for four coordinates $A\prime$, $B\prime$, $C\prime$, and $D\prime$ is on the right.

10. Repeat steps 7–9 until $I_C = I_B$.
11. Move I_A along line $\overline{A\prime D\prime}$ by an amount $= \frac{d_{ad}}{height_{original}}$.
12. Move I_B along line $\overline{B\prime C\prime}$ by an amount $= \frac{d_{bc}}{height_{original}}$.
13. Increment o_y.
14. Repeat steps 5–13 until $I_A = D\prime$ and $I_B = C\prime$.

This is feasible on the FPGA by using lookup tables to calculate sin, cos, and arctan for angle calculations. Besides that, it needs a relatively small number of multiplications, just two per pixel in the original image, and four per line in the original image. There is also a square root that is needed once per line, this can be implemented with either a look up table, or by using an iterative method of calculation.

3.6 VGA Write

4 External Components

5 List of Goals