Project Proposal
Augmented Reality Image Processing System

Logan P. Williams & José E. Cruz Serrallés

November 15, 2011

Abstract

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 a low-pass filter and a projective transformation to project an image onto the dimensions of the picture frame. 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.

1 Top-Level Block Diagram

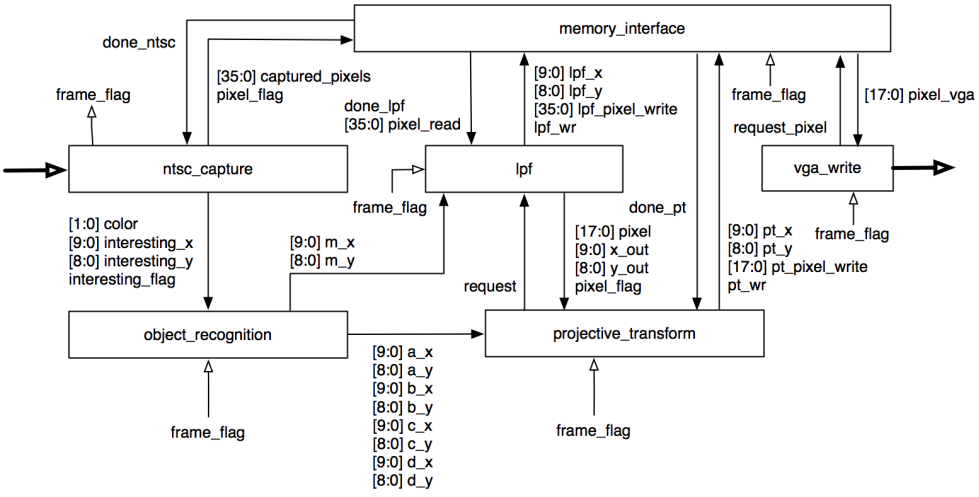

Figure 1: *The block diagram of the augmented reality system.*

2 Submodules

2.1 `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 gives pixels to `memory_interface` to write into ZBT memory.

There are two modifications. The first allows the module to capture and store color data in Y/Cr/Cb format. The second modification is added to support object recognition. When the `ntsc_capture` module sees a pixel of a color that matches the target (blue, green, red, and yellow), it sends information (the color and its X/Y location) to the `object_recognition` module. It also outputs a flag that goes high when the entire frame has been captured, and a new one is beginning. A more precise definition of the inputs and outputs, their width, and their name can be seen in the block diagram above for every module described below.

This module can be tested by connecting it to the `vga_write` module and ensuring that the video output is the image seen by the video camera.

2.2 `memory_interface` (José)

The `memory_interface` module handles the interaction between all of the other modules and the two ZBT Memory blocks, which house the four images that the modules use for capturing, displaying, and processing. Ideally, BRAM would have been used, but the number of pixels that we would like to store vastly exceeds BRAM capacity. Unlike BRAM, each ZBT Memory block can only handle one read or write operation per cycle, causing memory access to be the main bottleneck of our system. As such, we will store only the six most significant bits of each component of every pixel, allowing us to store two pixels per address and to reduce the number of memory accesses in our system by a factor of two. The number of required memory acceses per module and the distribution of the images in the RAM necessitates a minimum clock frequency of 50.7MHz, which is reasonable given the propagation delays of multipliers and other elements.

`memory_interface` will allocate two images per memory block. These four images will be (1) *capture*, the image being captured from NTSC; (2) *display*, the image being displayed in the VGA; (3) *processing*, the image that will be processed by `LPF` and projected by `projective_transform`; and (4) *next_display*, the image to which `projective_transform` will write and the next image that will be displayed. Every image refresh (1/30 seconds), the previous *next_display*, *display*, *capture*, and *processing* image locations will become the next *display*, *processing*, *next_display*, and *capture* image locations, respectively. These location shifts will be transparent to the other modules. Read and write requests from `vga_write` and `ntsc_capture` will be given priority over requests from other modules.

The inputs to `memory_interface` are (1) frame_flag, which signals when to shift; (2) two pixels from `ntsc_capture`; (3) two (x,y) pairs, one from `LPF` and one from `projective_transform`; (4) two pixels from `LPF`; (5) one pixel from `projective_transform`; and (6) request flags and (7) write flags from other submodules. The outputs from `memory_interface` are (1) done flags; (2) one pixel to `vga_write`; and (3) two pixels to `LPF`.

`memory_interface` will be tested in stages. Initially, basic read and write functionality will be assessed in simulation and then on the FPGA. Once we have written and read information from the ZBT RAM

successfully, we will attempt to write, read, and display an image. Finally, all of the logic pertaining to handling read and write requests from all the modules simulatenously will be written, tested using extensive testbenches, and finally tested on an FPGA with dummy modules. All of this testing should avoid help us avoid headaches during final integration.

2.3 `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 inputs (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 a new frame is beginning. It produces as output four sets of X/Y coordinates, one for 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 `LPF` and the `projective_transfom` module. This will require a divide module, which can be created with coregen. `object_recognition` can be tested with a simple test bench that provides some sample pixel locations, and tests to see if the module computes the center of mass correctly.

2.4 `LPF` (José)

The `LPF` module's sole purpose is to apply a lowpass filter (LPF) to the *processing* image so as to avoid aliasing when `projective_transform` skews the image. The following steps detail the operation of `LPF` every image refresh cycle (1/30 seconds): (1) Load the filter coefficients of a 1D LPF with cutoff frequency $\frac{\pi}{M_y}$. (2) Apply this filter to each column of *processing* and store each column once again in *processing*. (3) Load the filter coefficients of a 1D LPF with cutoff frequency $\frac{\pi}{M_x}$. (4) Apply this filter to each row of *processing* but instead feed the output pixels to `projective_transform`. (5) Wait for the next cycle. The image data will be buffered in BRAM, such that LPF accesses memory 1.5 times per pixel.

The inputs to `LPF` are (1) M_x and (2) M_y, the downsampling coefficients; (3) frame_flag, which signals when to start filtering; (4) the done pulse and (5) the pixels from `memory_interface`; and (5) the request from projective_transform. The outputs from `LPF` to `memory_interface` are (1) the write signal, (2) the pixels to be written, and the (3) (x,y) coordinates of the leftmost written pixel. The outputs from `LPF` to `projective_transform` are (1) the pixel flag, which signals when a new pixel is available; (2) the (x,y) coordinates of this new pixel; and (3) the new pixel.

The lowpass filters that will be used will be finite impulse response (FIR) Parks-McClellan filters. Most of the information contained in an image is contained in its phase. FIR filters were chosen because they can be made so as to have no effect on the phase of the image, preserving most of the information. Parks-McClellan filters were chosen because they are highly adaptable and easily calculated with MATLAB. The filter coefficients will be stored in BRAM for easy access. Because `FIR` will only be filtering the luminance component of each pixel, the order, N, of these filters will only be constrained by the number of multipliers on the FPGA and the number of multipliers used in other modules. The symmetry of these filters will be exploited, requiring only $\frac{N}{2}+1$ multiplications per pixel. We are aiming for filters of order 100, though filters of order 50 or greater will suffice.

Because `LPF` is used only to make the output look nice, `LPF` will be delegated to the end of the project. Given enough time, this module will be written and tested extensively with progressively more complicated testbenches. The initial testbench will apply to filter to an image with one white pixel, with all cutoff coefficients. The outputs of this testbench should match the coefficients in the BRAM. Once the module passes these tests, `LPF` will be used on more complicated images, and the output will be compared to the ideal output with MATLAB. In these testbenches, different memory access cases will be tested, as well.

2.5 `projective_transform` (Logan)

The inputs to `projective_transform` are (1) the pixel value last produced by `LPF`, (2) a flag signal held high for one clock signal when `LPF` has processed a new pixel, (3) the four coordinates of the corners of the frame provided by the `object_recognition` module, and (4) a signal when a new frame is beginning.

The outputs from `projective_transform` are (1) a request to `LPF` for a new pixel, (2) the X/Y coordinates of the transformed pixel, (3) the transformed pixel value, and (4) a flag indicating that a new pixel is to be written.

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 Figure 2, on the next page.

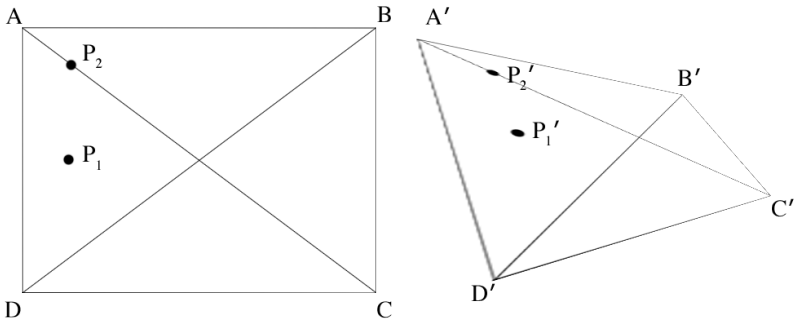

Figure 2: *A visual representation of the result of the* `projective_transform` *module. Input is on the left, a possible output, for four coordinates A′, B′, C′, and D′ is on the right.*

Mathematically, the algorithm works as follows:

1. Calculate the distance of line $\overline{A'D'}$ and assign it to d_{ad}.
2. Do the same for $\overline{B'C'}$ and assign it to d_{bc}.
3. Create two "iterator points," point I_A and I_B initially located at A' and B'.
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.
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 algorithm 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, which will be implemented with an iterative algorithm.

The `projective_transform` module can be tested by creating a test bench that provides a series of test pixels as input. The output generated in ModelSim by `projective_transform` can then be compared with the output generated by a MATLAB implementation of the algorithm described above.

2.6 `vga_write` (José)

The `vga_write` section is straightforward and will basically be a clone of the VGA code used in Lab 2 (Pong). Essentially, the output will be refreshed at a rate of 60Hz and output at a resolution of 640x480 pixels. This module will read the pixel values from displayed image and assign it based on the *hcount* and `vcount` variables, which will be incremented accordingly. The clock signal used for this module will be synthesized with the ISE's toolkit. This module will be tested by loading a standardized image to memory and verifying whether `vga_write` displays this image properly on the monitor.

3 External Components

We will be using two standard external components: a video camera that provides NTSC composite video out, to be provided by the 6.111 staff, and a VGA display, available in the 6.111 lab.

4 Project Deadlines

11-11-2011 Finalized Block Diagram

11-18-2011 First drafts of `projective_transform` and `memory_interface` written

11-22-2011 First drafts of `object_recognition`, `LPF`, `vga_write`, and `ntsc_capture` first drafts written; `projective_transform` and `memory_interface` fully tested

11-28-2011 `ntsc_capture` and vga_write fully tested; start of basic integration

11-31-2011 `object_recognition` and `LPF` fully tested; start of full integration

12-05-2011 Full integration complete

12-12-2011 Final report due