HERA: 360° Image Warping Surveillance System

Design Document

Team Members
Bryan Gwin
David Lariviere
DJ Park
Michael Verbalis

Table of Contents

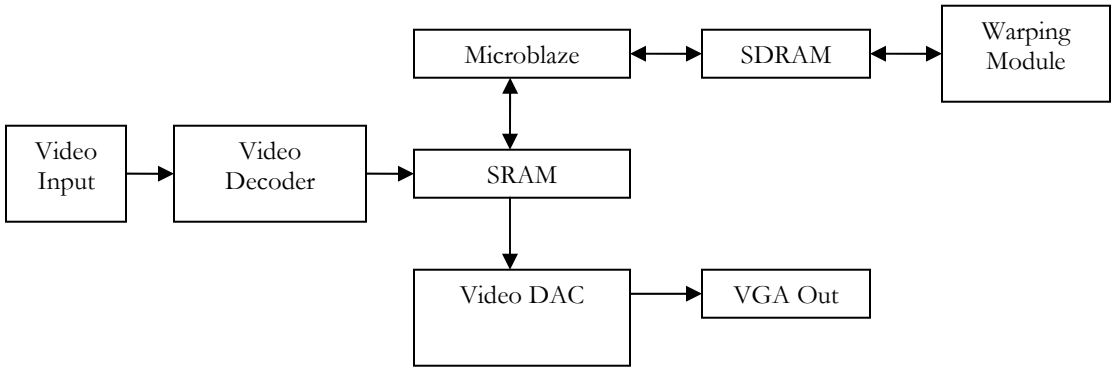
1.0 System Design

1.1 Overview

The goal of this project is to take a real-time 360° video stream into the video input of the XESS XSB-300E board and to un-warp a portion of it in order to display a proper image onto a monitor. Due to memory restrictions of the XSB board, we will be outputting the video in a 4 bit grayscale mode.

1.2 Block Diagram

The following block diagram describes the current flow of data in this system:



2.0 Components

2.1 Camera System

In order to capture the initial 360° video stream, we will be using a standard 360° lens mounted onto a PC165C Color ExView CMount Video Camera. This camera is in the family of high performance security cameras and has extremely sharp resolution (480 x 480) for a camera its size.

2.2 Video-In

We will be using the on-board Phillips SAA7114-H video decoder chip to handle the analog video signal obtained from the camera. The chip is capable of digitizing any NTSC, SECAM, or PAL video signal and scaling it to an appropriate frame size. The chip also allows for precise control over the luminance, chrominance, huge, saturation, etc. of the input video through numerous controls.

Our system will be taking roughly a 240 x 240 slice of the input frame to pass into the SRAM. The slice coordinates will be calculated by the warping mechanism.

2.3 BRAM

We will be using the on-chip Block RAM on the FPGA that features 512 x 8 bits of dual-ported memory per block. This is .5K each and there are 16 such blocks on the board = 8K max.

These on-chip RAMs are very fast and are thus useful for storing the image data to and from the SRAM. Half of this RAM memory will be used to store incoming lines of image data from the video decoder and the other half will be used to store the unwarped image data read from the SRAM.

2.4 SRAM

We will be using the Toshiba TC55V16256J SRAM that features 256K x 16 bits of memory = 512K

The SRAM is used to store the video input before warping. We're taking 240x240x4bit (grey-scale) = 230K of the memory for this purpose. The image data will be continuously loaded (real-time) from the video decoder, and will be continuously read-out to the warping module. The warping module will then write back another 230K of unwarped image data to the other half of the SRAM. Thus we are using a total of 460K out of 512K.

2.5 SDRAM

We will be using the Samsung K4S281632 SDRAM that features 8M x 16 bits of memory = 128M. We will use the Pipelined SDRAM controller module from: *http://www.xess.com/ho03000.html*.

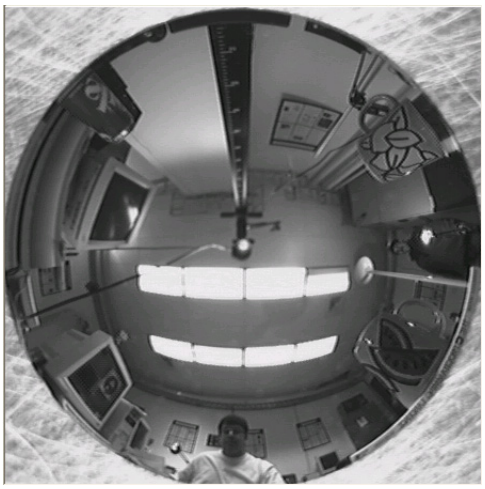
The amount of memory needed by the warping module is 256x256x16 = 1M which is too much to be stored anywhere but the SDRAM. Thus, the mapping table will be stored here. Although this RAM is slower, it will not be read and written as much as the video data on the SRAM. Furthermore, the mapping table can be read in bursts, greatly increasing the speed of the SDRAM. (A non-pipelined read/write cycle take 7/5 clock cycles, whereas pipelining the data in bursts only takes 2/1 clock cycles with some (small) initial setup cycles per row.)

2.6 Warping Module

The input signal consists of a video stream from a 360 degree catadioptric lens (see image on the next page). The purpose of warping is to take a portion of the 360 degree image and unwarp it into a standard perspective-view image

representing the image that would have been captured from a standard camera taken in the corresponding direction.

The computational cost of mapping input pixels to corresponding output pixels is rather large and is therefore not possible in real-time. Therefore the mapping of corresponding pixels will be calculated once (for the desired view) before hand and stored in a lookup table.



360° Input Image

Unwarped Perspective View

The warping process will use a "nearest-neighbor" filter, meaning that each output pixel will be calculated by using the value of the closest corresponding input pixel. The look-up table will map each output pixel to the corresponding input pixel.

$$f(x_{out}, y_{out}) \dashrightarrow (x_{in}, y_{in})$$

The lookup-table therefore controls the portion of the 360 degree scene the viewer will see on the monitor. It will be generated by software running on the microblaze and then written to the SDRAM. Because the data required during unwarping will be sequentially required, pipelining will be possible when reading in the required portion of the look-up table. The user will be able to change the current view by sending keyboard commands sent over the serial terminal. This will cause the system to generate a new lookup-table mapping a different portion of the screen.

The actual input pixels determined by the mapping function will in most cases be accessed non-sequentially, and therefore it will be stored in SRAM. The warping module will also be responsible for generating bounds for the required area of the input image needed so that only the required portions of the image are stored in SRAM.

Assuming an input image of 256x256 pixels, 16 bits will be required for each pixel, consisting of 8 bits for the x and y values.

Please see Appendix for an example C program that generates unwarped images.

2.7 Video DAC

We will be using the onboard Texas Intruments THS8133B Digital-To-Analog Converter that features a triple 10-bit Digital to Analog Converter (DAC) to translate and deliver each video frame to a VGA display.

Using the OPB Bus, two video outputs (240x240 warped and un-warped images) will be read from the SRAM and outputted onto the screen. The final display will be 640x480 and unused pixels will be zeroed.

3.0 Future Considerations

At this stage of the project, we realize that there are many unknown factors. We will be able to better understand these as we continue working.

1. The placement of the warping module is not yet finalized because there is no way to know the exact timing until implementation.
2. As the speed of the warping is not yet known, we may have to drop frames from the input and only process every nth frame.
3. Depending on the speed of the SDRAM, we may be able to put all of the video data directly onto it rather than using the SRAM. This could potentially allow us to use color data as well.

4.0 Appendix

```
/*****
Modified by David Lariviere
Homework 5
W4165– Pixel Processing, Fall 2005
*****/

/* ============================================================================
 * IMPROC: Image Processing Software Package
 * Copyright (C) 2001 by George Wolberg
 *
 * omniview.skel - Skeleton program for generating perspective views
 *                 from omnivideo images.
 *
 * Written by: George Wolberg, 2001
 * ============================================================================
 */

#include "IP.h"
#include "pgm.h"

#define SGN(A)          ((A) > 0 ? 1 : ((A) < 0 ? -1 : 0 ))
#define FLOOR(A)        ((int) (A))
#define CEILING(A)      ((A)==FLOOR(A) ? FLOOR(A) : SGN(A)+FLOOR(A))
#define CLAMP(A,L,H)  ((A)<=(L) ? (L) : (A)<=(H) ? (A) : (H))

typedef double  vectorS[3];

/* Dot product: compute scalar A dot B */
#define vectorDot(A, B)  ((A[0]*B[0]) + (A[1]*B[1]) + (A[2]*B[2]))

/* Norm: compute length of vector A */
#define vectorNorm(A)   (sqrt( vectorDot(A,A) ))

/* Normalize vector A into unit vector */
#define vectorNormalize(A)      \
        {                       \
        double len;             \
        len = vectorNorm(A);    \
        if(len == 0) len = 1;   \
        A[0] /= len;            \
        A[1] /= len;            \
        A[2] /= len;            \
        }

/* Cross product: C = A x B */
#define vectorCross(A, B, C)    \
        {                       \
        vectorS T;              \
        T[0] = A[1]*B[2] - A[2]*B[1];       \
        T[1] = A[2]*B[0] - A[0]*B[2];       \
        T[2] = A[0]*B[1] - A[1]*B[0];       \
        memcpy((char*) C, (char*) T, sizeof(vectorS));    \
        }
```

```
#define NN       0
#define LINEAR1
#define CUBIC  2

extern void omniview(imageP, int, int, int, int, int, int, imageP);

main(int argc, char **argv)
{
  char *inputFile, *outputFile;
  int newWidth, newHeight, func, dimensions;
  imageP inImg, outImg;
  int xc, yc, cop;
  int r, i;
  int numDivisions=12;
  double theta;
  if (argc!=9) {
          printf("Invalid arguements\n");
          printf("Usage: omniview input_img new_width new_height method xc yc cop
output_img\n");
          exit(-1);
  }

  //usage: omniview input_img new_width new_height method xc yc cop output_img
  inputFile = argv[1];
  newWidth = atoi(argv[2]);
  newHeight = atoi(argv[3]);
  func = atoi(argv[4]); //0=>box, 1=>triangle, 2=>cubic
  xc = atoi(argv[5]);
  yc = atoi(argv[6]);
  cop = atoi(argv[7]);
  outputFile = argv[8];

  //read in source image
  inImg = IP_readImage(inputFile);

  xc = 223;
  yc = 223;
  r = 111;

  for (i=0; i<numDivisions; i++) {
          //allocate destination output image
          outImg = IP_allocImage(newWidth, newHeight, sizeof(uchar));
          theta = (double)(i) / (double) numDivisions * 2.0 * 3.14159;
          //resample the input image and put into outputImg
          omniview(inImg, xc+(r*cos(theta)), yc+(r*sin(theta)), cop, newWidth, newHeight, func,
outImg);

          sprintf(outputFile, "output%d.pgm", i);
          //save image
          //IP_saveImage(outImg, outputFile);
          IP_savePGMImage(outImg, outputFile);
   }
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
 * cubicConv:
 *
 * Cubic convolution filter.
 * A is preset here to -1
 */
double cubicConv(double t) {
        double A;
        double t2, t3;
        A = -1;

        if(t < 0) t = -t;
        t2 = t  * t;
        t3 = t2 * t;

        if(t < 1.0) return((A+2)*t3 - (A+3)*t2 + 1);
        if(t < 2.0) return(A*(t3 - 5*t2 + 8*t - 4));
        return(0.0);
}



/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * omniview:
 *
 * Compute perspective view from the omnivideo (parabolic) image in I1.
 * Output image I2 consists of the scene projected on the viewport.
 * The viewport dimensions are w x h. Its normal axis N passes through
 * the center of projection (origin) and the 3D point that corresponds
 * to the selected (xc,yc) on the omnivideo image.
 * The viewport lies a distance cop from the origin of the world coord sys.
 * filter specifies the intrp method: 0=NN; 1=bilinear.
 * NOTE: we assume that I1 has been cropped so that it is a square image
 * of dimensions 2r*2r, where r is the radius of the parabolic image.
 */
void
omniview(I1, xc, yc, cop, w, h, filter, I2)
imageP  I1, I2;
int         xc, yc, cop, w, h, filter;
{
        int         i, j, x, y, ix, iy, ww, hh;
        uchar    *ip, *op, *pp, bf[4];
        double   xx, yy, zc, r, rr, rmax, phi, theta;
        double   dx, dy;
        vectorS  U, V, N, P, S;
        double weight; //used during debugging
        double sum;
        /* allocate output buffer */
        //.....

        /* init input and output buffer pointers */
        ip = (uchar *) I1->image;
        op = (uchar *) I2->image;

        /* radius of parabolic mirror (in pixels) */
        rr = (ww = I1->width) / 2;
```

```
//height
hh = I1->height;

/* move center (xc,yc) to origin and find zc, the position on mirror */
xc = xc - rr;
yc = yc - rr;
zc = (rr*rr - (xc*xc + yc*yc)) / (2*rr);

/* N is unit vector along VPN: passes through (0,0,0) and (xc,yc,zc) */
N[0] = xc;
N[1] = yc;
N[2] = zc;
vectorNormalize(N);

/* U is unit vector of the perpendicular to N projected on z=0 plane */
U[0] = -N[1];
U[1] =  N[0];
U[2] =    0 ;
vectorNormalize(U);

/* V is the cross product of U and N: V = U x N */
vectorCross(U, N, V);

/* use P to store the world coords of viewport points */
P[2] = cop;

/* init rmax to account for neighboring pixels needed by filter */
rmax = (rr-filter) * (rr-filter);

/* project all viewport points into the parabolic image */
for(y=0; y<h; y++) {
        P[1] = y - h/2;
        for(x=0; x<w; x++) {
                P[0] = x - w/2;

                /* transform world coord point into UVN coord sys */
                S[0] = U[0]*P[0] + V[0]*P[1] + N[0]*P[2];
                S[1] = U[1]*P[0] + V[1]*P[1] + N[1]*P[2];
                S[2] = U[2]*P[0] + V[2]*P[1] + N[2]*P[2];
                vectorNormalize(S);

                /* spherical coords for vector through current pixel */
                phi   = atan2(S[1], S[0]);
                theta = acos (S[2]);

                /* radius of intersection on parabolic mirror */
                r  = 2 * rr * (1-cos(theta)) / (1-cos(2*theta));

                /* convert from polar to world coordinates */
                xx = r * sin(theta) * cos(phi);
                yy = r * sin(theta) * sin(phi);

                /* check if image point lies inside parabolic image */
                if(xx*xx + yy*yy < rmax) {
                        ix = xx + rr;
                        iy = yy + rr;
```

```
                                        pp = &ip[iy*ww + ix];

                                        switch(filter) {
                                        case NN:
                                                /* nearest neighbor */
                                                *op++ = *pp;
                                                break;
                                        case LINEAR:
                                                /* bilinear intrp */
                                                dx = (xx + rr) - (double) FLOOR(xx + rr);
                                                dy = (yy + rr) - (double) FLOOR(yy + rr);
                                                *op++ = ((pp[0]*(1.0-dx) + pp[1]*dx)*(1-dy) +
(*(pp+ww)*(1.0-dx) + *(pp+ww+1)*dx)*dy);
                                                break;
                                        case CUBIC:
                                                /* cubic convolution */
                                                dx = (xx + rr) - (double) FLOOR(xx + rr);
                                                dy = (yy + rr) - (double) FLOOR(yy + rr);
                                                //calculate cubic interpolation for rows
                                                sum = 0;
                                                for (i=0; i<5; i++) {
                                                        for (j=0; j<5; j++) {
                                                                weight = cubicConv(dx-2.0 + (double) j)
* cubicConv(dy-2.0 + (double)i);
                                                                sum += ip[CLAMP((iy-2+i),0,hh)*ww +
CLAMP((ix-2+j),0,ww)] * weight;
                                                        }
                                                }
                                                *op++ = CLAMP(sum, 0, 255);
                                                //*op++ = CLAMP(cubRows[1],0,255);

                                                //*op++ = CLAMP(cubRows[0] * cubicConv(dy - .0) +
cubRows[1] * cubicConv(dy) + cubRows[2] * cubicConv(dy+1.0), 0, 255);
                                                //*op++ = .....
                                                break;
                                        }
                                } else  *op++ = 0;
                        }
                }
}
```