REAL-TIME FACE DETECTION AND TRACKING

A Design Project Report
Presented to the School of Electrical and Computer Engineering of Cornell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by
Thu-Thao Nguyen
MEng Field Advisor: Bruce Robert Land
Degree Date: December 2012

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Real-Time Face Detection and Tracking

Author: Thu-Thao Nguyen

Abstract:

Face detection and tracking has been an important and active research field because it offers many applications, especially in video surveillance, biometrics, or video coding. The goal of this project was to implement a real-time system on an FPGA board to detect and track a human's face. The face detection algorithm involved color-based skin segmentation and image filtering. The face location was determined by calculating the centroid of the detected region. A software version of the algorithm was independently implemented and tested on still pictures in MATLAB. Although the transition from MATLAB to Verilog was not as smooth as expected, experimental results proved the accuracy and effectiveness of the real-time system, even under varying conditions of lights, facial poses and skin colors. All calculation of the hardware implementation was done in real time with minimal computational effort, thus suitable for power-limited applications.

Executive Summary

The area of face detection and tracking plays an important role in many applications such as video surveillance, biometrics, or video coding. Many different methodologies have been proposed in literature and can mostly be categorized as featured-based, appearance-based, or color-based. Among these approaches, color-based face detection algorithm was found to be most efficient as it required low computational cost while being robust to variations in lighting, facial expressions, and skin colors.

The goal of this project was to create an FPGA system to detect and track a human's face in real time. The overall setup included the Verilog program, an Altera DE2 board, a camera, and a VGA monitor. The face detection algorithm implemented here was based on skin detection and image filtering. After the face region was detected, its location was determined by calculating the centroid of neighboring skin pixels.

A software-based algorithm was independently developed and examined in MATLAB to evaluate its performance and verify its effectiveness. However, it was infeasible to implement the same algorithm in Verilog due to the limitations of the language (i.e. calling a function recursively was not allowed). Hence, several stages of the algorithm were modified. Experimental results proved the accuracy and effectiveness of the hardware real-time implementation as the algorithm was able to handle varying types of input video frame. All calculation was performed in real time.

Although the system can be furthered improved to obtain better results, overall the project was a success as it enabled any inputted face to be accurately detected and tracked.

Table of Contents

Introduction

Face detection and tracking is the process of determining whether or not a face is present in an image. Unlike face recognition—which distinguishes different human faces, face detection only indicates whether or not a face is present in an image. In addition, face tracking determines the exact location of the face. Face detection and tracking has been an active research area for a long time because it is the initial important step in many different applications, such as video surveillance, face recognition, image enhancement, video coding, and energy conservation. The applicability of face detection in energy conservation is not as obvious as in other applications. However, it is interesting to learn how a face detection and tracking system allows power and energy to be saved. Suppose one is watching a television and working on other tasks simultaneously. The face detection system is for checking whether or not the person is looking directly at the TV. If the person is not directly looking at the TV within some time period (i.e. 15 minutes), the TV's brightness is reduced to save energy. When the person turns back to look at the TV, the TV's brightness can be increased back to original. In addition, if the person looks away for too long (i.e. more than one hour), then the TV will be automatically turned off.

Different approaches to detect and track human faces—including feature-based, appearance-based, and color-based have been actively researched and published in literature. The feature-based approach detects a human's face based on human facial features—such as eyes and nose. Because of its complexity, this method requires lots of computing and memory resources. Although compared to other methods this one gives higher accuracy rate, it is not suitable for power-limited devices. Hence, a color-based algorithm is more reasonable for applications that require low computational effort. In general, each method has its own advantages and disadvantages. More complex algorithm typically gives very high accuracy rate but also requires lots of computing resources.

Design and Implementation

Algorithm

General design stages are illustrated in Figure 1.

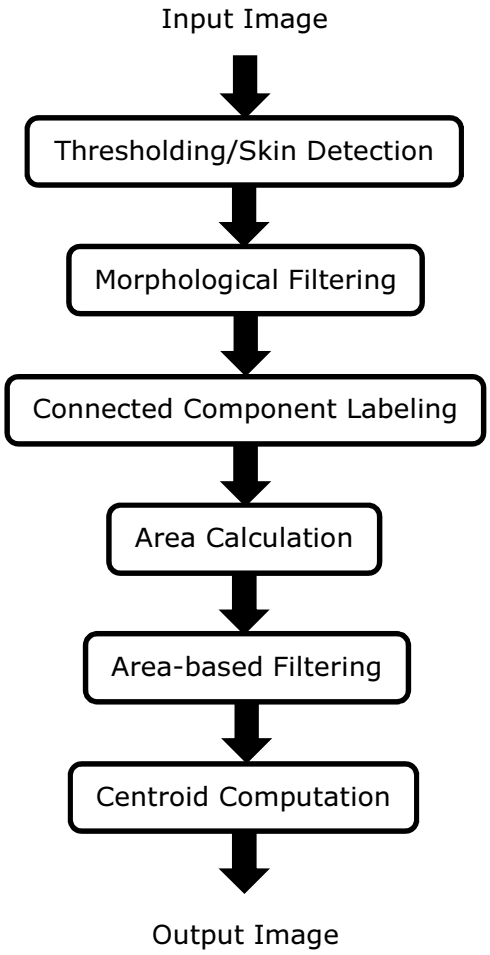

Figure 1 - Software Algorithm

The skin detection algorithm here was derived from the method describe in [1]. Color segmentation has been proved to be an effective method to detect face regions due to its low computational requirements and ease of implementation. Compared to the featured-based method, the color-based algorithm required very little training.

First, the original image was converted to a different color space, namely modified YUV. Then the skin pixels were segmented based on the appropriate U range. Morphological filtering was applied to reduce false positives. Then each connected region of detected pixels in the image was labeled. The area of each labeled region was computed and an area-based

filtering was applied. Only regions with large area were considered face regions. The centroid of each face region was also computed to show its location.

Modified YUV Color Space

Converting the skin pixel information to the modified YUV color space would be more advantageous since human skin tones tend to fall within a certain range of chrominance values (i.e. U-V component), regardless of the skin type. The conversion equations are shown as follows [1].

$$Y = \frac{R+2G+B}{4}$$

$$U = R - G$$

$$V = B - G$$

These equations allowed thresholding to work independently of skin color intensity.

45 34 30	#2D221E
60 46 40	#3C2E28
75 57 50	#4B3932
90 69 60	#5A453C
105 80 70	#695046
120 92 80	#785C50
135 103 90	#87675A
150 114 100	#967264
165 126 110	#A57E6E
180 138 120	#B48A78
195 149 130	#C39582
210 161 140	#D2A18C
225 172 150	#E1AC96
240 184 160	#F0B8A0
255 195 170	#FFC3AA
255 206 180	#FFCEB4
255 218 190	#FFDABE
255 229 200	#FFE5C8

Figure 2 - Different skin tone samples [1]

Thresholding/Skin Detection

After skin pixels were converted to the modified YUV space, the skin pixels can be segmented based on the following experimented threshold.

$$10 < U < 74$$

$$-40 < V < 11$$

As seen in Figure 2, the blue channel had the least contribution to human skin color. Additionally, according to [2], leaving out the blue channel would have little impact on thresholding and skin filtering. This also implies the insignificance of the V component in the YUV format. Therefore, the skin detection algorithm using here was based on the U component only. Applying the suggested threshold for the U component would produce a binary image with raw segmentation result, as depicted in Figure 3.

Figure 3 - Result after thresholding

Morphological Filtering `(imerode, imfill)`

Realistically, there are so many other objects that have color similar to the skin color. As seen in Figure 3, there are lots of false positives present in the raw segmentation result. Applying morphological filtering—including erosion and hole filling would, firstly, reduce the background noise and, secondly, fill in missing pixels of the detected face regions, as illustrated in Figure 4. MATLAB provided built-in functions—`imerode` and `imfill` for these two operations.

```
outp = imerode(inp, strel('square', 3));
```

The command `imerode` erodes the input image `inp` using a square of size 3 as a structuring element and returns the eroded image `outp`. This operation removed any group of pixels that had size smaller than the structuring element's.

```
outp = imfill(inp, 'holes');
```

The command `imfill` fills holes in the binary input image `inp` and produces the output image `outp`. Applying this operation allowed the missing pixels of the detected face regions to be filled in. Thus, it made each face region appear as one connected region.

Figure 4 - Result after morphological filtering

Connected Component Labeling and Area Calculation `(bwlabel, regionprops)`

After each group of detected pixels became one connected region, connected component labeling algorithm was applied. This process labeled each connected region with a number, allowing us to distinguish between different detected regions. The built-in function `bwlabel` for this operation was available in MATLAB. In general, there are two main methods to label connected regions in a binary image—known as recursive and sequential algorithms.

```
[L, n] = bwlabel(inp);
```

The command `bwlabel` labels connected components in the input image `inp` and returns a matrix `L` of the same size as `inp`. `L` contains labels for all connected regions in `inp`. `n` contains the number of connected objects found in `inp`.

The command `regionprops` can be used to extract different properties, including area and centroid, of each labeled region in the label matrix obtained from `bwlabel`.

```
face_region = regionprops(L, 'Area');
face_area = [face_region.Area];
```

The two commands above performed two tasks (1) extract the area information of each labeled region (2) store the areas of all the labeled regions in the array `face_area` in the order of their labels. For instance `face_area(1) = 102` would mean the area of the connected component with label "1" is 102 pixels.

Area-based Filtering `(find, ismember)`

Note that morphological filtering only removed some background noise, but not all. Filtering detected regions based on their areas would successfully remove all background noise and any skin region that was not likely to be a face. This was done based on the assumption that human faces are of similar size and have largest area compared to other skin regions, especially the hands. Therefore, to be considered a face region, a connected group of skin pixels need to have an area of at least 26% of the largest area. This number was obtained from experiments on training images. Therefore, many regions of false positives could be removed in this stage, as depicted in Figure 5.

```
face_idx = find(face_area > (.26)*max(face_area));
face_shown = ismember(L, face_idx);
```

These two commands performed the following tasks (1) look for the connected regions whose areas were of 26% of the largest area and store their corresponding indices in `face_idx` (2) output the image `face_shown` that contained the connected regions found in (1).

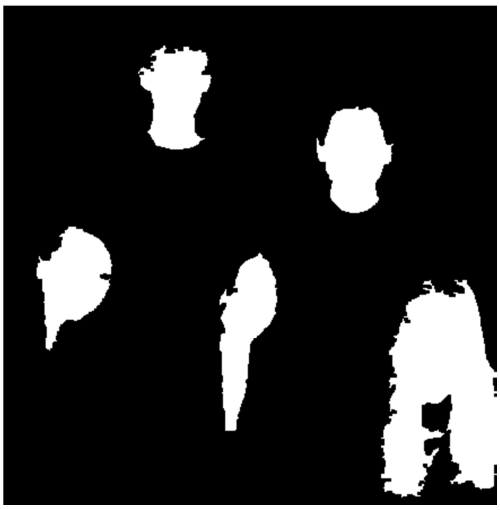

Figure 5 - Result after area-based filtering

Centroid Computation

The final stage was to determine face location. The centroid of each connected labeled face region can be calculated by averaging the sum of X coordinates and Y coordinates separately. The centroid of each face region in Figure 6 is denoted by the blue asterisk. Here the centroid of each connected region was extracted using `regionprops`.

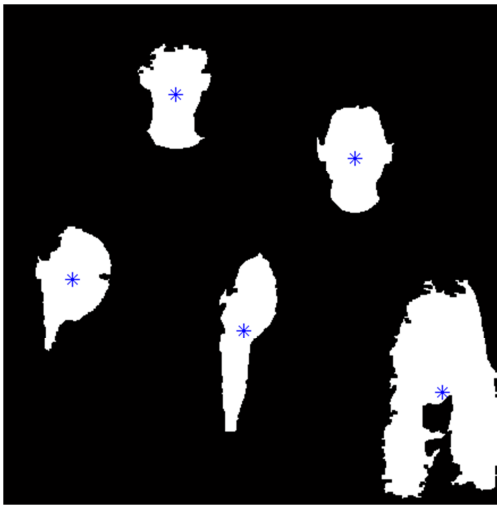

Figure 6 - Result after calculating centroid

MATLAB—Algorithm Testing

Before the algorithm was developed in Verilog, it was implemented and tested on still images in MATLAB to verify its functionality, as illustrated from Figure 3 to Figure 6. The Image Processing Toolbox provided in MATLAB allowed the process of developing and testing the algorithm to be more efficient. Furthermore, verifying the accuracy of the detection algorithm on still pictures provided fair results.

However, the transition from MATLAB to Verilog coding was not as smooth as expected due to the limitations of the Verilog language. Specifically, morphological filtering was modified and extended to spatial and temporal filtering. In addition, connected component labeling was not implemented in Verilog as this language does not allow calling a function recursively. A modified detection algorithm will be discussed later.

Verilog—Hardware Implementation

Overall the FPGA system was setup as illustrated in Figure 7.

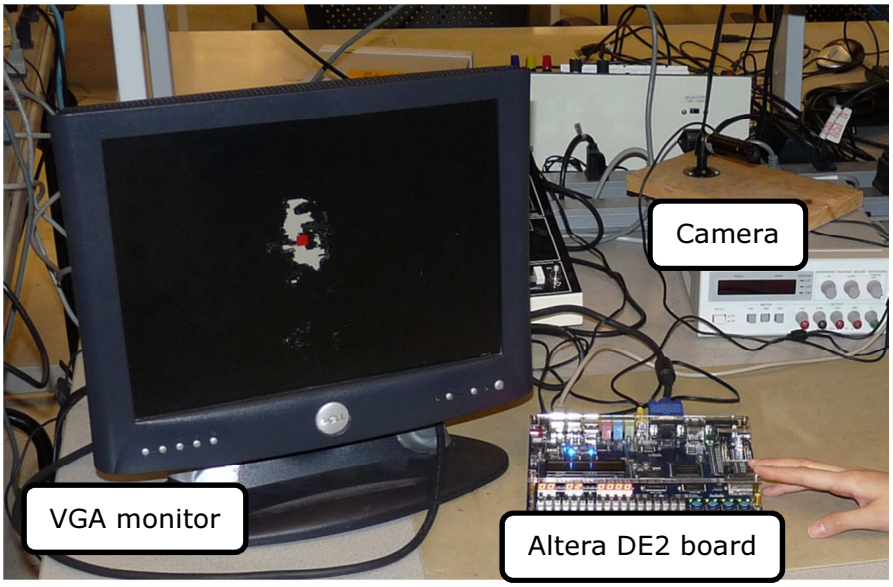

Figure 7 - Hardware Setup

Each current video frame was captured by the camera and sent to the FPGA's decoder chip via a composite video cable. After the video signal was processed in different modules in Verilog, the final output passed through the VGA driver to be displayed on the VGA monitor.

The hardware algorithm was modified as shown in Figure 8.

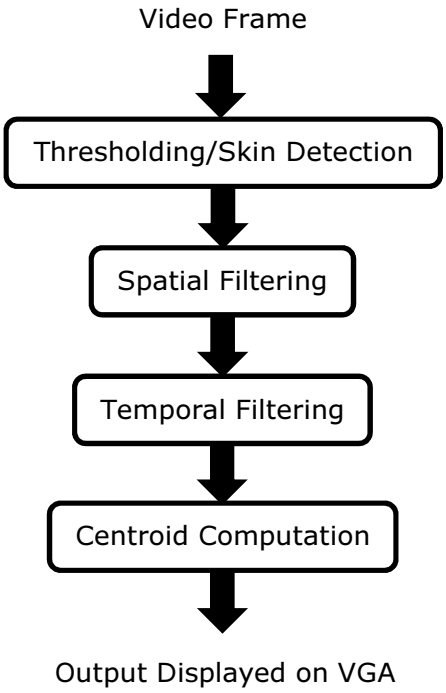

Figure 8 - Hardware Algorithm

Thresholding

Since 10-bit color was used in Verilog, adjusting the aforementioned U range yields

$$40 < U < 296$$

In this step, each input video frame was converted to a "binary image" showing the segmented raw result.

Spatial Filtering

This step was similar to the erosion operation used in the software algorithm. However, the structuring element used here did not have any particular shape. Instead, for every pixel *p*, its neighboring pixels in a 9x9 neighborhood were checked. If more than 75% of its neighbors were skin pixels, *p* was also a skin pixel. Otherwise *p* was a non-skin pixel. This allowed most background noise to be removed because usually noise scattered randomly through space, as shown in Figure 9. In Figure 10, because *p* only had 4 neighboring pixels categorized as skin, *p* was concluded to be a non-skin pixel and, thus, converted to a background pixel.

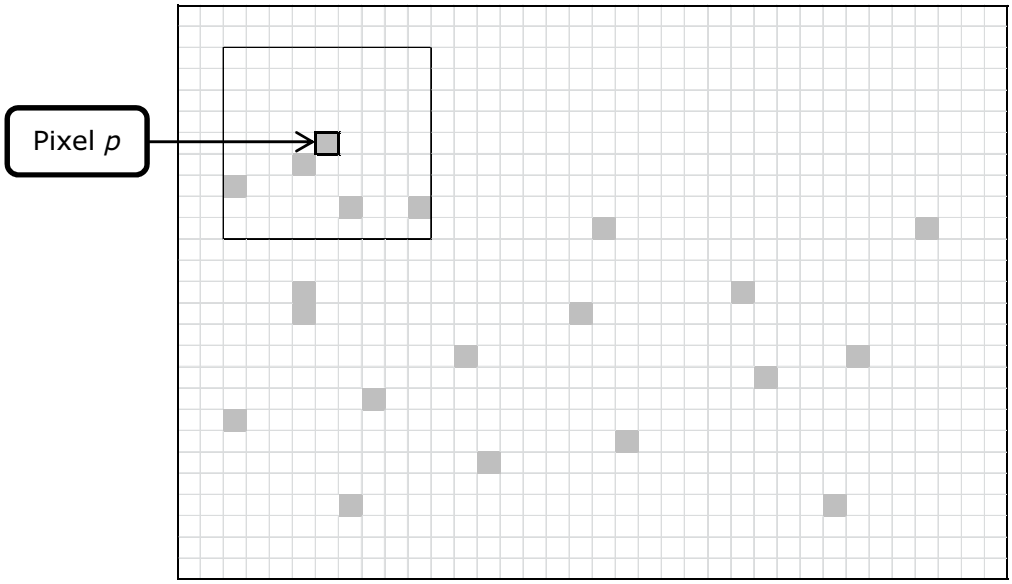

Figure 9 - Example of spatial filtering for a pixel *p*—before filtering

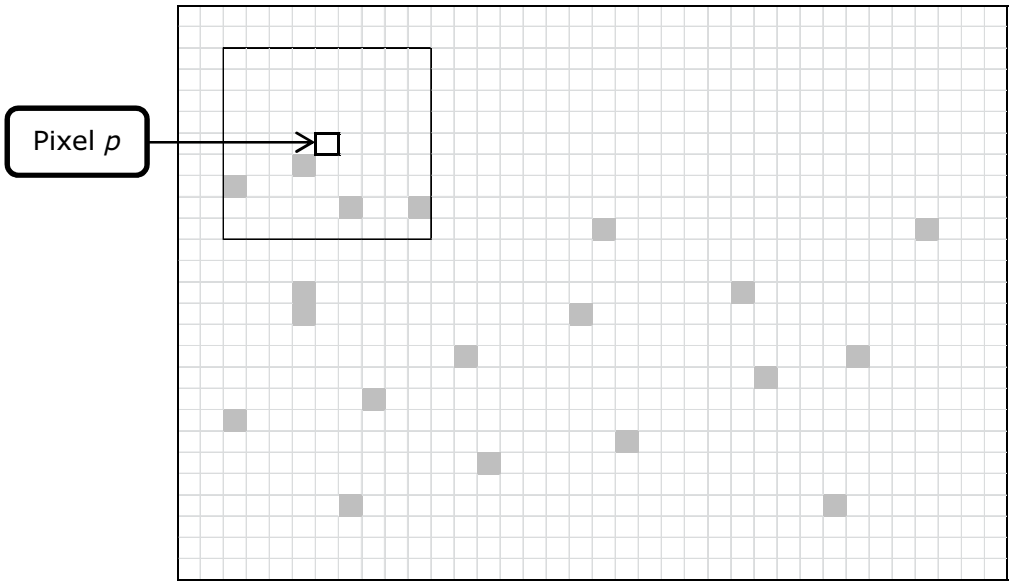

Figure 10 - Example of spatial filtering for a pixel *p*—after filtering

To examine the neighbors around a pixel, their values needed to be stored. Therefore, ten shift registers were created to buffer the values of ten consecutive rows in each frame. As seen in Figure 11, each register was 640-bit long to hold the binary values of 640 pixels in a row. Each bit in data_reg1 was updated according to the X coordinate. For instance, when the X coordinate was 2, data_reg1[2] was updated according to the result of thresholding from the previous stage. Thus, data_reg1 was updated every clock cycle. After all the bits of data_reg1 were updated, its entire value was shifted to data_reg2. Thus, other registers (from data_reg2 to data_reg10) were only updated when the X coordinate was 0. Values of data_reg2 to data_reg10 were used to examine a pixel's neighborhood.

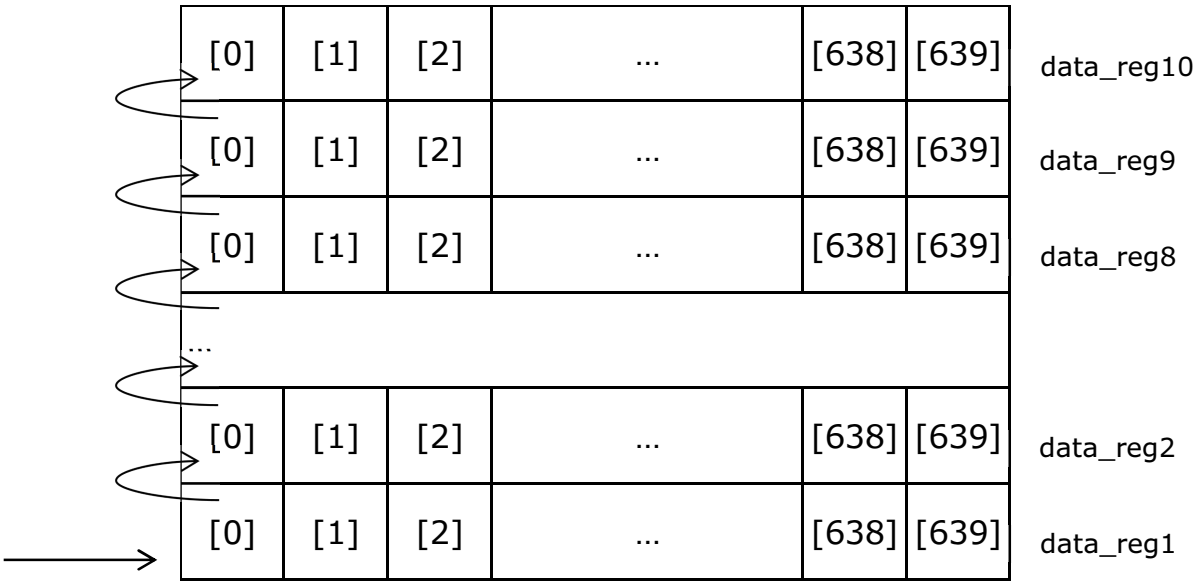

Figure 11 - Ten shift registers for ten consecutive rows

There was a trade-off between the number of shift registers being used (i.e. the size of the neighborhood) and the performance of the spatial filter. A larger neighborhood required more registers to be used but, at the same time, allowed more noise to be removed.

Temporal Filtering

Even small changes in lighting could cause flickering and made the result displayed on the VGA screen less stable. Applying temporal filtering allowed flickering to be reduced significantly. The idea of designing such a filter was borrowed from the project "Real-Time Cartoonifier" (see References for more information of this project). The temporal filter was based on the following equation.

avg_out = (3/4) avg_in + (1/4) data

data: filtered result obtained from the previous stage of a pixel, namely *p*, in current frame

avg_in: average value of *p* from previous frame

avg_out: average value of *p* in current frame

This is approximately equal to averaging four consecutive frames over time. To ease the computational effort, the equation above can be re-written as

avg_out = avg_in – (1/4) avg_in + (1/4) data

avg_out = avg_in – avg_in >> 2 + data >> 2

The filtered result of a pixel in this stage was determined based on its average value (i.e. avg_out). If its average value was greater than 0.06 (number obtained from experiments), the pixel was considered skin. Otherwise, the pixel was non-skin. Figure 12 and Figure 13

illustrate the process of temporal filtering for two pixels *p*1 and *p*2. In both examples, pixel *p*1 and *p*2 are truly skin pixels. However, the results before filtering were unstable due to light variations. The temporal filter smoothed the output and, thus, reduced flicker significantly.

frame	i	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9	i+10	i+11	i+12	i+13	i+14	i+15	i+16	i+17
before filtering	0	1	1	1	0	1	1	1	1	0	0	1	1	1	0	1	0	0
average value	0.00	0.25	0.44	0.58	0.43	0.58	0.68	0.76	0.82	0.62	0.46	0.60	0.70	0.77	0.58	0.68	0.51	0.39
after filtering	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 12 – Example of temporal filtering for a pixel *p*1

frame	i	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9	i+10	i+11	i+12	i+13	i+14	i+15	i+16	i+17
before filtering	1	0	1	0	1	1	0	1	0	1	0	1	0	1	1	1	0	0
average value	0.00	0.00	0.25	0.19	0.39	0.54	0.41	0.56	0.42	0.56	0.42	0.57	0.42	0.57	0.68	0.76	0.57	0.43
after filtering	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 13 – Example of temporal filtering for a pixel *p*2

Centroid Computation

Finally, centroid was computed to locate the face region. Because connected component labeling was not implemented as initially planned, it was infeasible to calculate the centroid for each face region separately. This limited the number of faces to be detected to two as maximum. First assume that only one face was present. Therefore, its centroid would just be the centroid of all detected pixels, as shown in Figure 14. Note that this calculation would only be correct if one face was present.

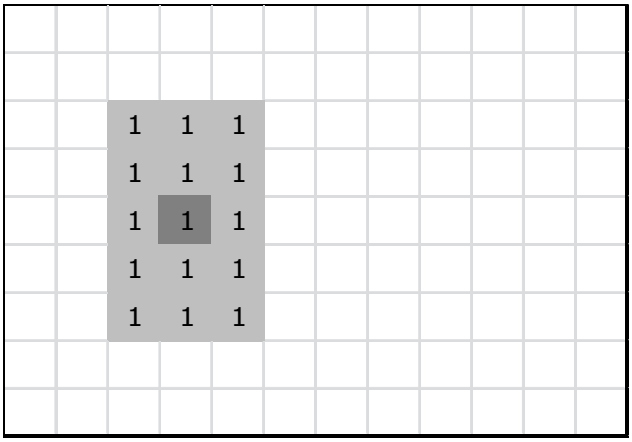

Figure 14 - Centroid of all detected pixels

Although the pixels of one face region might not be connected (and labeled) as originally planned, simply calculating the centroid of all detected pixels still gave a good estimate for the face location, as shown in Figure 15. Since area-based filtering was also not applied (due to the lack of connected component labeling), other skin regions—mostly the hands

were not entirely removed. However, even if the hands were present, calculating the centroid of all detected pixels still allowed us to locate the face region. This was a reasonable estimate because, compared to the face area, the area of the hand/hands was much smaller.

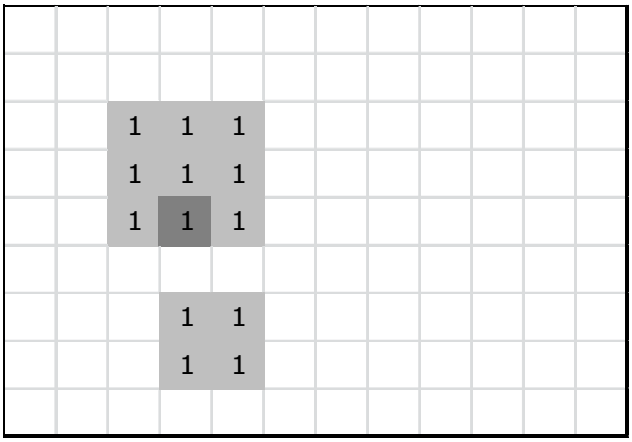

Figure 15 - Centroid of all detected pixels—one person

However, when there were two faces present, calculating the centroid of all detected pixels would only track the location between two faces, rather than track each face separately. To separately track each face in a two-person frame, additional steps were required. First the neighboring pixels around the centroid were checked to see if they were skin pixels. If they were, it meant the centroid accurately located the face region. However, if the neighboring pixels of the centroid were not skin pixels, it meant the centroid was somewhere in the background located between two detected face regions, as described in Figure 16.

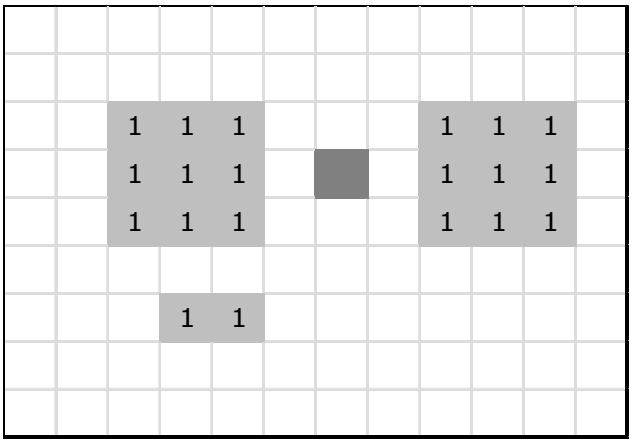

Figure 16 - Centroid of all detected pixels—two people

To solve this problem, the video frame was split into two according to where the centroid was, as represented in Figure 17. Figure 18 shows the separate calculation for the centroid

of each detected region. This technique was done based on the assumption that people typically sit side by side.

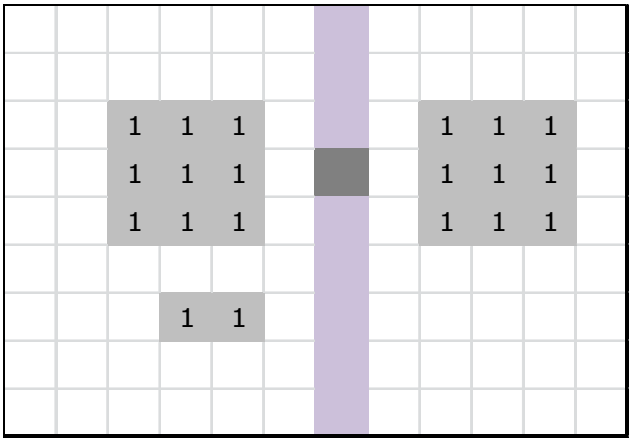

Figure 17 - Dividing video frame according to centroid location

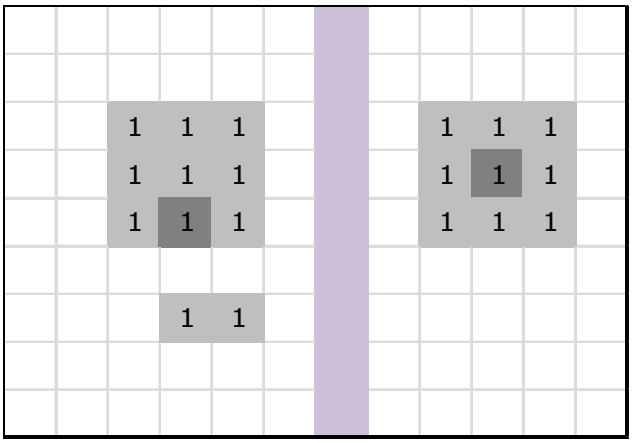

Figure 18 - Centroid of each detected face

Obtaining the centroid of each face region allowed us to locate the face of each person present in a two-person video frame.

To show how a face was tracked, a small box was drawn around the centroid. The box moved according to the movement of the face. However, if the face moved too fast, the movement of the box might become less stable. Applying temporal filtering here allowed the box to move smoothly. The implementation of the temporal filter here was slightly different from the one shown previously.

$Y_n = (1 - \alpha)X_n + \alpha Y_{n-1}$

X_n: current input

Y_n: current output

Y_{n-1}: previous output

The input here was the location of the centroid before filtering. What this equation meant was, with α being close to 1, current output would be more dependent on previous output than on current input. This prevented the centroid box from moving too fast when there was an abrupt change in the movement of a person's face.

Results

Performance

The final result was a complete system that was capable to detect and track faces of at most two people in real time. Although it was not able to track each face separately when there were three people or more, it could still detect the presence of their faces. Experiments also showed that different light settings did not significantly alter the final results. Furthermore, the system was able to ignore background noise very well—mostly came from light reflection. When there were objects that had color similar to skin color, both spatial and temporal filtering helped erode these detected regions, therefore reducing the number of false positives.

Sample Results and Analysis

When there was no one (Figure 19, Figure 20, Figure 21, and Figure 22)

- From Figure 20 to Figure 21 it can be seen that many false positives (mostly background noise) were removed.
- From Figure 21 to Figure 22, the scarf was eroded. Also because the number of detected pixel count was below the threshold, the centroid of these pixels did not appear at all. This implied no face was detected and tracked.

Still image taken from a book (Figure 23 and Figure 24)

- Even in a real-time system, a person's face from a still image can still be detected (and tracked—if we moved the book manually).

Presence of one person (Figure 25, Figure 26, Figure 27, and Figure 28)

- The red letters on the hoodie were not completely removed but it was eventually eroded after every stage. Although there were still false positives in the final result, the centroid still correctly indicated the face location.

Presence of one person (with scarf) (Figure 29, Figure 30, Figure 31, and Figure 32)

- Although the scarf was not completely eroded, it did not significantly impact the final detection and tracking result.

Presence of two people (Figure 33, Figure 34, Figure 35, and Figure 36)

- When there were two people, the system was still able to detect and track their faces. The location of each face was accurately determined.

Figure 19 - When there was no one—natural

Figure 20 - When there was no one—skin detection

Figure 21 - When there was no one—spatial filtering

Figure 22 - When there was no one—temporal filtering + centroid computation

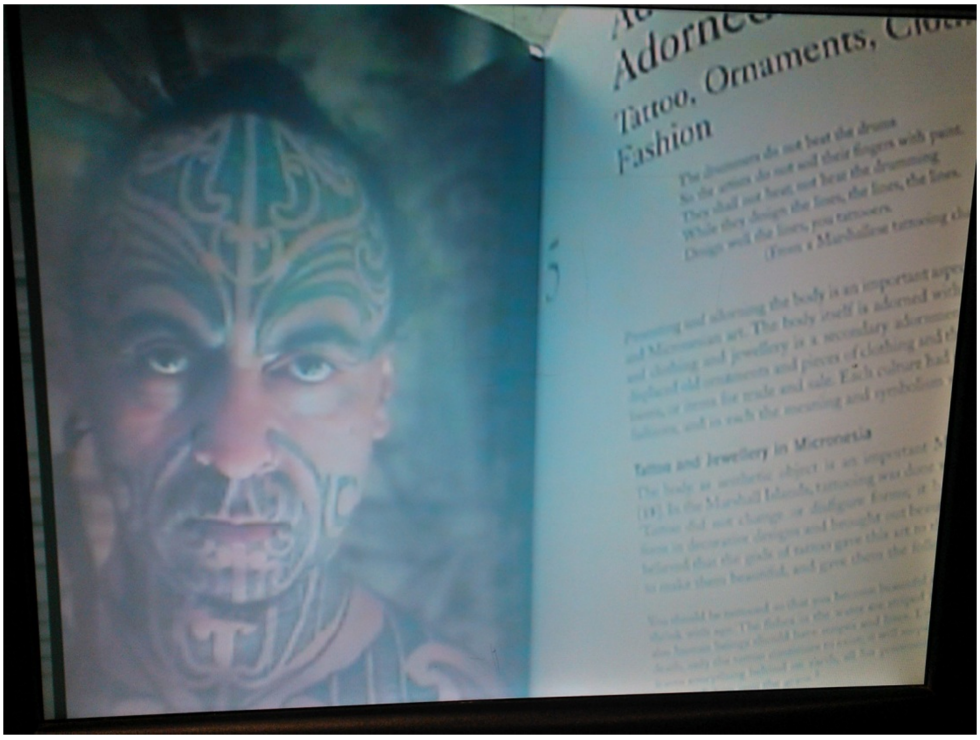

Figure 23 – Still image taken from a book—natural

Figure 24 - Still image taken from a book—final result

Figure 25 - Presence of one person—natural

Figure 26 - Presence of one person—skin detection

Figure 27 - Presence of one person—spatial filtering

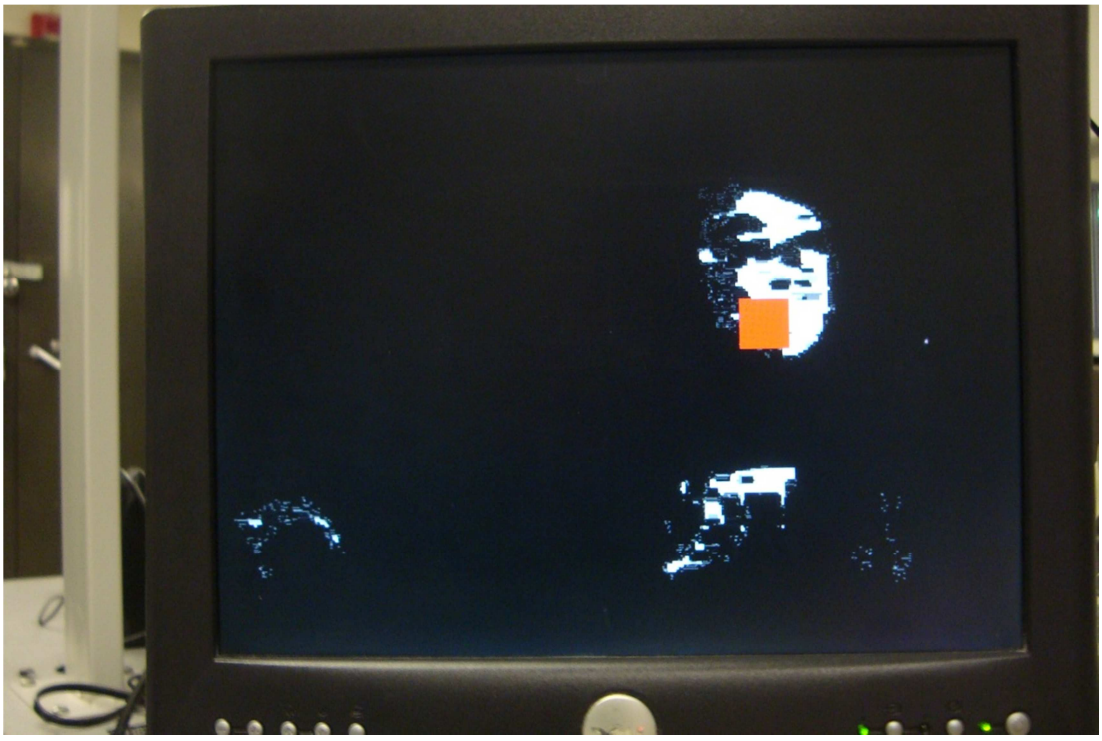

Figure 28 - Presence of one person—temporal filtering + centroid computation

Figure 29 - Presence of one person (with scarf)—natural

Figure 30 - Presence of one person (with scarf)—skin detection

Figure 31 - Presence of one person (with scarf)—spatial filtering

Figure 32 - Presence of one person (with scarf)—temporal filtering + centroid computation

Figure 33 - Presence of two people—natural

Figure 34 - Presence of two people—skin detection

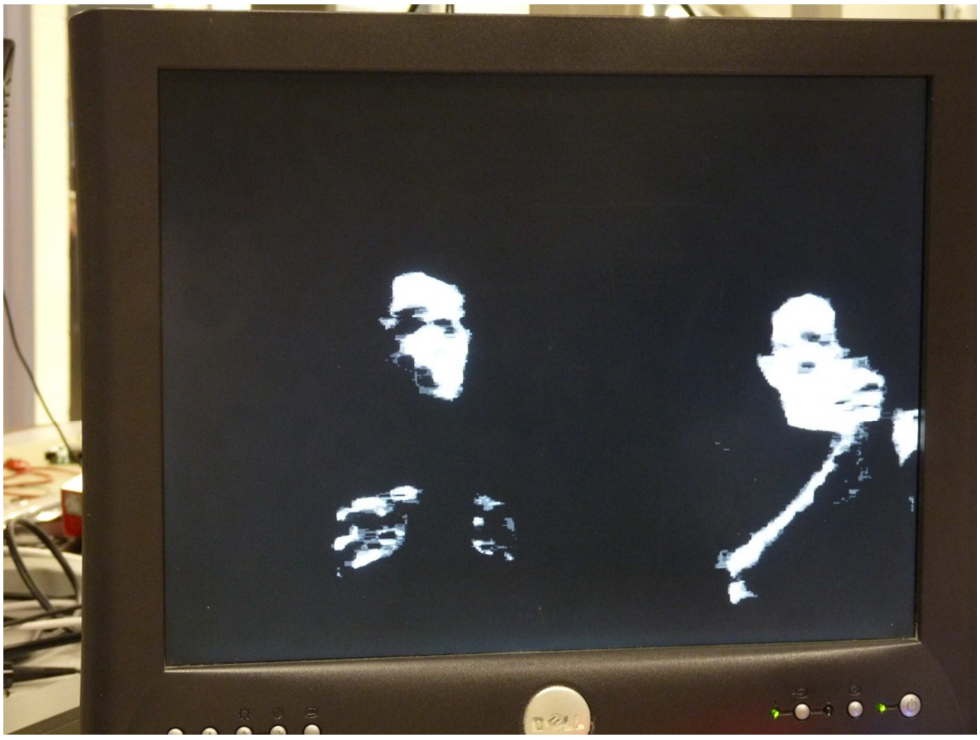

Figure 35 - Presence of two people—spatial filtering

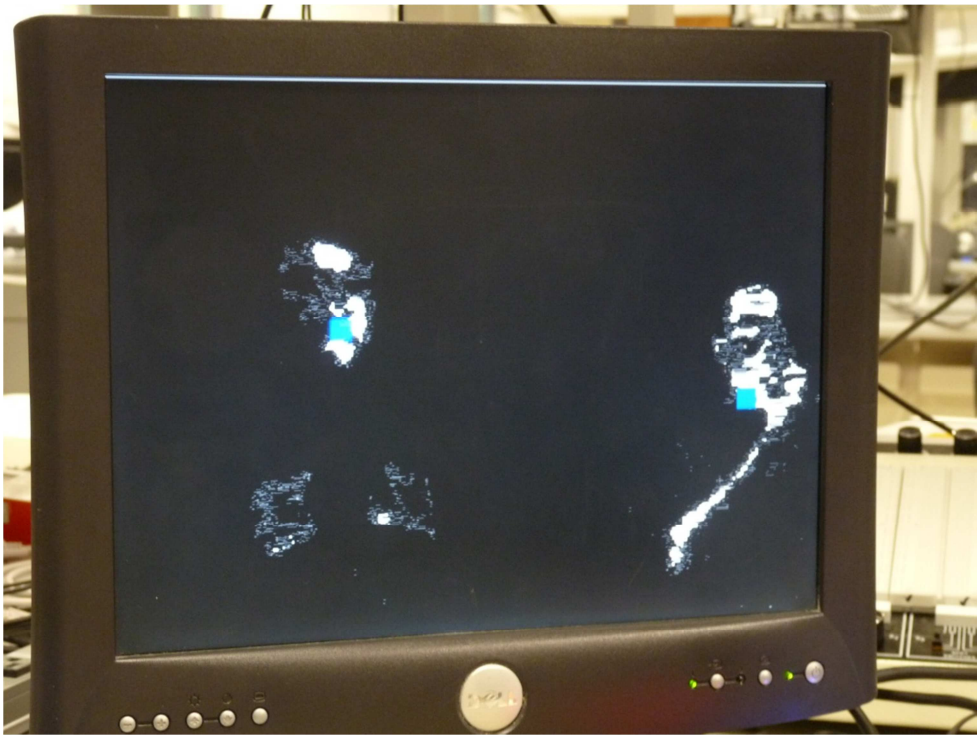

Figure 36 - Presence of two people—temporal filtering + centroid computation

Speed

A clock of 27 MHz was used for the face detection and tracking algorithm. Since the timing was synchronized with the VGA clock, the VGA display was able to update within the time gap between drawing two consecutive frames. Therefore, the camera was able to detect and track people' faces in real time.

Accuracy

Error seemed to occur only when there was a transition from one person to two people or vice versa in the video frame. Within the lab setting, noise was very minimal and did not alter the results. As long as a person was in the camera's view, his face would be accurately detected and tracked. His distance relative to the camera did not affect the result.

Limitations

In the presence of three or more people, the system could only detect the faces but failed at tracking them.

Conclusion

In this project, the goal of implementing a hardware system to detect and track human faces in real time was achieved. A software implementation of the algorithm was examined in MATLAB to verify its accuracy. Although the transition from software to hardware required some modification to the original algorithm, the initial goal was still accomplished. The face detection algorithm was derived from a skin detection method. Face tracking was achieved by computing the centroid of each detected region, although it only worked in the presence of at most two people. Different types of filter were applied to avoid flickering and stabilize the output displayed on the VGA screen. The system was proved to work in real time with no lagging and under varying conditions of facial expressions, skin tones, and lighting.

Acknowledgements

I would like to thank Professor Bruce Land who was the faculty advisor for my design project. This project would have been impossible without his weekly help and guidance.

References

[1] M. Ooi, "Hardware Implementation for Face Detection on Xilinx Virtex-II FPGA Using the Reversible Component Transformation Color Space," in *Third IEEE International Workshop on Electronic Design, Test and Applications*, Washington, DC, 2006.

[2] S. Paschalakis and M. Bober, "A Low Cost FPGA System for High Speed Face Detection and Tracking," in *Proc. IEEE International Conference on Field-Programmable Technology*, Tokyo, Japan, 2003.

[3] [Online]. Available: http://www.jwp.se/files/skintone.jpg. [Accessed December 2012].

[4] [Online]. Available: http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2010/kaf42_jay29_teg25/teg25_jay29_kaf42/index.html. [Accessed October 2012].

Appendices

MATLAB code

```
close all
clear all
clc

%%Author: ThaoNguyen
%%Face detection and tracking

%%Read original image and get its size%%
face_original = imread('image08.png');
size_img = size(face_original);
r = 1; g = 2; b = 3;
y = 1; u = 2; v = 3;

%%YUV Conversion%%
face_yuv = face_original;
for i = 1:size_img(1)
    for j = 1:size_img(2)
        face_yuv(i, j, y) = (face_original(i, j, r) + 2*face_original(i, j,
g) + face_original(i, j, b)) / 4;
        face_yuv(i, j, u) = face_original(i, j, r) - face_original(i, j, g);
        face_yuv(i, j, v) = face_original(i, j, b) - face_original(i, j, g);
    end
end

%%Skin segmentation%%
face_detected = face_original;
for i = 1:size_img(1)
    for j = 1:size_img(2)
        %U range was found based on experiments
        if face_yuv(i, j, u) > 20 && face_yuv(i, j, u) < 74
            %Set suspected face regions to 1
            face_detected(i, j, r) = 255;
            face_detected(i, j, g) = 255;
            face_detected(i, j, b) = 255;
        else
            %Set non-face regions to 0
            face_detected(i, j, r) = 0;
            face_detected(i, j, g) = 0;
            face_detected(i, j, b) = 0;
        end
    end
end

%%Convert image into the BW format; the image itself stays the same, only the
format is changed%%
face_detected = im2bw(face_detected);

%%Erode noise%%
face_imerode = imerode(face_detected, strel('square', 3));
```

```
%%Fill in holes to get fully connected face regions%%
face_imfill = imfill(face_imerode, 'holes');

%%Label each connected region in the BW image%%
[L, n] = bwlabel(face_imfill); %n gives us #connected objects

face_region = regionprops(L, 'Area', 'BoundingBox');
face_area = [face_region.Area]; %contains areas of all the filled regions

%%Filter out regions whose areas are less than 26% largest area (supposedly a
face area)%%
face_idx = find(face_area > (.26)*max(face_area)); %only shows indices of
regions that are faces
face_shown = ismember(L, face_idx);

%%Face areas vs. box areas%%
fprintf('Ratio between face area and box area \n');
for n = 1:length(face_idx)
    idx = face_idx(n);
    area_face(n) = face_region(idx).Area;
    area_box(n) =
face_region(idx).BoundingBox(3)*face_region(idx).BoundingBox(4);
    ratio(n) = area_face/area_box;
end
ratio

%%Compute the coordinates of each face region%%
for n = 1:length(face_idx)
    idx = face_idx(n);
    face_region(idx).BoundingBox;
    xmin = round(face_region(idx).BoundingBox(1));
    ymin = round(face_region(idx).BoundingBox(2));
    xmax =
round(face_region(idx).BoundingBox(1)+face_region(idx).BoundingBox(3));
    ymax =
round(face_region(idx).BoundingBox(2)+face_region(idx).BoundingBox(4));
    %%Draw a box around each face found%%
    for i = xmin:xmax
        face_original(ymin, i, r) = 255;
        face_original(ymin, i, g) = 0;
        face_original(ymin, i, b) = 0;
    end

    for i = xmin:xmax
        face_original(ymax, i, r) = 255;
        face_original(ymax, i, g) = 0;
        face_original(ymax, i, b) = 0;
    end

    for j = ymin:ymax
        face_original(j, xmin, r) = 255;
        face_original(j, xmin, g) = 0;
        face_original(j, xmin, b) = 0;
    end
```

```
    for j = ymin:ymax
        face_original(j, xmax, r) = 255;
        face_original(j, xmax, g) = 0;
        face_original(j, xmax, b) = 0;
    end
end

%imshow(face_original), title ('Original Image')
imshow(face_detected), title ('Raw Segmentation Result')
figure, imshow(face_imerode), title ('Eroded Result')
figure, imshow(face_imfill), title ('Filled Regions')
figure, imshow(face_shown), title ('Final Result')

%%Compute centroids of the face regions%%
s  = regionprops(face_shown, 'Centroid');
centroids = cat(1, s.Centroid); %cat puts all the s.Centroid values into a
matrix
figure, imshow(face_original), title('Detection Result'), xlabel (['#faces
found: ', num2str(length(face_idx))])
hold on
plot(centroids(:,1), centroids(:,2), 'b*', 'MarkerSize', 10)
hold off
```

Verilog code

```
//Real-Time Face Detection and Tracking
//by ThaoNguyen

reg [9:0] fltr, fltr2, fltr3;
reg [9:0] raw_R, raw_G, raw_B;
reg [9:0] fltr2_R, fltr2_G, fltr2_B;
reg [9:0] fltr3_R, fltr3_G, fltr3_B;
reg [15:0] fltr_reg;

wire [9:0] avg_in, avg_out, avg2;

reg [639:0] data_reg1, data_reg2, data_reg3, data_reg4, data_reg5, data_reg6,
data_reg7, data_reg8, data_reg9, data_reg10, data_reg11;

wire[9:0] VGA_Red, VGA_Green, VGA_Blue;

reg [9:0] VGA_X1, VGA_Y1;

reg [9:0] avgX, avgY, avgX_lpf, avgY_lpf;
reg [29:0] sumX, sumY;
reg [18:0] cntr;

reg [9:0] avgX_L, avgY_L, avgX_L2, avgY_L2;
reg [29:0] sumX_L, sumY_L;
reg [18:0] cntr_L;

reg [9:0] avgX_R, avgY_R, avgX_R2, avgY_R2;
reg [29:0] sumX_R, sumY_R;
reg [18:0] cntr_R;

assign we = VGA_X1[0]; //write enable for SRAM, active low

// SRAM control
```

```
assign SRAM_ADDR = {VGA_X1[9:1], VGA_Y1[9:1]};
assign SRAM_DQ = VGA_X1[0] ? avg_out : 16'hzzzz;

assign SRAM_UB_N = 0;
assign SRAM_LB_N = 0;
assign SRAM_CE_N = 0;
assign SRAM_WE_N = VGA_X1[0] ? 1'b0 : 1'b1;
assign SRAM_OE_N = 0;

assign avg_in = VGA_X1[0] ? avg_in : SRAM_DQ;                                  // 10 bits
assign avg_out = avg_in - (avg_in >> 2) + (fltr2 >> 2);
assign avg2 = avg_out << 4;

assign sw_17 = DPDT_SW[17];
assign sw_16 = DPDT_SW[16];
assign sw_15 = DPDT_SW[15];
assign reset = KEY[0];

// ********** Display Options
************************************************************************ //
assign VGA_Red =           (sw_17 && sw_16)    ? mRed :                          //
original data
                                               (~sw_17 && sw_16) ? raw_R  :
            // raw result
                                               (sw_17 && ~sw_16) ? fltr3_R: fltr2_R;
// fltr3 displays final result, fltr2 displays result after spatial filtering

assign VGA_Green =  (sw_17 && sw_16)    ? mGreen     :
                                               (~sw_17 && sw_16)   ? raw_G      :
                                               (sw_17 && ~sw_16) ? fltr3_G: fltr2_G;

assign VGA_Blue =          (sw_17 && sw_16)    ? mBlue      :
                                               (~sw_17 && sw_16)   ? raw_B      :
                                               (sw_17 && ~sw_16) ? fltr3_B: fltr2_B;
//
********************************************************************************************
************** //

always@(posedge OSC_27)
begin

// ********** Registering X, Y coordinates
********************************************************** //
      VGA_X1 <= VGA_X;
      VGA_Y1 <= VGA_Y;
//
********************************************************************************************
************** //

      if (~KEY[1])
      begin
            data_reg1 <= 'd0;

            sumX <= 30'b0;
            sumY <= 30'b0;
            cntr <= 19'b0;
            sumX_L <= 30'b0;
            sumY_L <= 30'b0;
            cntr_L <= 19'b0;
            sumX_R <= 30'b0;
            sumY_R <= 30'b0;
            cntr_R <= 19'b0;
```

```
            avgX_lpf <= avgX;
            avgY_lpf <= avgY;
            avgX_L2 <= avgX_L;
            avgY_L2 <= avgY_L;
            avgX_R2 <= avgX_R;
            avgY_R2 <= avgY_R;
            fltr_reg <= 16'd0;
      end

      else
      begin

            // ********** Temporal Filtering
************************************************************************ //
            if ((VGA_X1 < avgX_lpf + 10'd5) && (VGA_X1 > avgX_lpf - 10'd5) &&
                  (VGA_Y1 < avgY_lpf - 10'd5) && (VGA_Y1 > 10'd5)) begin
                  fltr_reg <= fltr_reg + fltr3[0];
            end
            else if ((VGA_X1 == 10'd600) && (VGA_Y1 == 10'd400)) begin
                  fltr_reg <= 16'd0;
            end

            if (fltr_reg > 16'd50) begin
                  if (avg2 > 10'b1110111111) begin // can also try b1110110111
                        fltr3 <= 10'h3FF;
                        fltr3_R <= 10'h3FF;
                        fltr3_G <= 10'h3FF;
                        fltr3_B <= 10'h3FF;
                        // Draw centroid
                        if (cntr > 19'd500) begin // Set the threshold so when
#pixels is too small, nothing will be detected
                              if ((VGA_X1 < avgX_lpf + 10'd20) && (VGA_X1 >
avgX_lpf - 10'd20) &&
                                    (VGA_Y1 < avgY_lpf + 10'd20) && (VGA_Y1 >
avgY_lpf - 10'd20)) begin
                                    fltr3_R <= 10'h3FF;
                                    fltr3_G <= 10'h0;
                                    fltr3_B <= 10'h0;
                              end
                        end
                  end
                  else begin
                        fltr3 <= 10'h0;
                        fltr3_R <= 10'h0;
                        fltr3_G <= 10'h0;
                        fltr3_B <= 10'h0;
                        if (cntr > 19'd500) begin
                              if ((VGA_X1 < avgX_lpf + 10'd20) && (VGA_X1 >
avgX_lpf - 10'd20) &&
                                    (VGA_Y1 < avgY_lpf + 10'd20) && (VGA_Y1 >
avgY_lpf - 10'd20)) begin
                                    fltr3_R <= 10'h3FF;
                                    fltr3_G <= 10'h0;
                                    fltr3_B <= 10'h0;
                              end
                        end
                  end
            end
            else begin
                  if (avg2 > 10'b1110111111) begin //before: b1110111111, (2)
b1110110111
                        fltr3 <= 10'h3FF;
                        fltr3_R <= 10'h3FF;
```

```
                        fltr3_G <= 10'h3FF;
                        fltr3_B <= 10'h3FF;
                        // Draw centroid
                        if (cntr > 19'd500) begin // Set the threshold so when
#pixels is too small, nothing will be detected
                              if ( ((VGA_X1 < avgX_R2 + 10'd10) && (VGA_X1 >
avgX_R2 - 10'd10) &&
                                    (VGA_Y1 < avgY_R2 + 10'd10) && (VGA_Y1 >
avgY_R2 - 10'd10)) ||
                                   ((VGA_X1 < avgX_L2 + 10'd10) && (VGA_X1 >
avgX_L2 - 10'd10) &&
                                    (VGA_Y1 < avgY_L2 + 10'd10) && (VGA_Y1 >
avgY_L2 - 10'd10)) ) begin
                                    fltr3_R <= 10'h0;
                                    fltr3_G <= 10'h0;
                                    fltr3_B <= 10'h3FF;
                              end
                        end
                  end
                  else begin
                        fltr3 <= 10'h0;
                        fltr3_R <= 10'h0;
                        fltr3_G <= 10'h0;
                        fltr3_B <= 10'h0;
                        if (cntr > 19'd500) begin
                              if ( ((VGA_X1 < avgX_R2 + 10'd10) && (VGA_X1 >
avgX_R2 - 10'd10) &&
                                    (VGA_Y1 < avgY_R2 + 10'd10) && (VGA_Y1 >
avgY_R2 - 10'd10)) ||
                                   ((VGA_X1 < avgX_L2 + 10'd10) && (VGA_X1 >
avgX_L2 - 10'd10) &&
                                    (VGA_Y1 < avgY_L2 + 10'd10) && (VGA_Y1 >
avgY_L2 - 10'd10)) ) begin
                                    fltr3_R <= 10'h0;
                                    fltr3_G <= 10'h0;
                                    fltr3_B <= 10'h3FF;
                              end
                        end
                  end
            end
            //
*****************************************************************************
************** //

            // ********** Computing centroid for all detected pixels
****************************************** //
            if ((VGA_X1 > 10'd20) && (VGA_X1 < 10'd620) &&
                  (VGA_Y1 > 10'd20) && (VGA_Y1 < 10'd460)) begin
                  if (fltr3 == 10'h3FF) begin
                        sumX <= sumX + VGA_X1;
                        sumY <= sumY + VGA_Y1;
                        cntr <= cntr + 19'b1;
                  end
            end

            if ((VGA_X1 == 10'd2) && (VGA_Y1 == 10'd478)) begin
                  avgX <= sumX / cntr;
                  avgY <= sumY / cntr;
                  avgX_lpf <= avgX_lpf - (avgX_lpf >> 'd2) + (avgX >> 'd2);
                  avgY_lpf <= avgY_lpf - (avgY_lpf >> 'd2) + (avgY >> 'd2);
                  sumX <= 30'b0;
                  sumY <= 30'b0;
                  cntr <= 19'b0;
```

```
            end
            //
*****************************************************************************************
************** //

            // ********** Computing centroid for left halved frame
********************************************** //
            if ((VGA_X1 > 10'd20) && (VGA_X1 < avgX_lpf - 10'd10) &&
                    (VGA_Y1 > 10'd20) && (VGA_Y1 < 10'd460)) begin
                  if (fltr3 == 10'h3FF) begin
                        sumX_L <= sumX_L + VGA_X1;
                        sumY_L <= sumY_L + VGA_Y1;
                        cntr_L <= cntr_L + 19'b1;
                  end
            end

            if ((VGA_X1 == 10'd20) && (VGA_Y1 == 10'd478)) begin
                  avgX_L <= sumX_L / cntr_L;
                  avgY_L <= sumY_L / cntr_L;
                  avgX_L2 <= avgX_L2 - (avgX_L2 >> 'd2) + (avgX_L >> 'd2);
                  avgY_L2 <= avgY_L2 - (avgY_L2 >> 'd2) + (avgY_L >> 'd2);
                  sumX_L <= 30'b0;
                  sumY_L <= 30'b0;
                  cntr_L <= 19'b0;
            end
            //
*****************************************************************************************
************** //

            // ********** Computing centroid for right halved frame
********************************************** //
            if ((VGA_X1 > avgX_lpf + 10'd10) && (VGA_X1 < 10'd620) &&
                    (VGA_Y1 > 10'd20) && (VGA_Y1 < 10'd460)) begin
                  if (fltr3 == 10'h3FF) begin
                        sumX_R <= sumX_R + VGA_X1;
                        sumY_R <= sumY_R + VGA_Y1;
                        cntr_R <= cntr_R + 19'b1;
                  end
            end

            if ((VGA_X1 == 10'd621) && (VGA_Y1 == 10'd478)) begin
                  avgX_R <= sumX_R / cntr_R;
                  avgY_R <= sumY_R / cntr_R;
                  avgX_R2 <= avgX_R2 - (avgX_R2 >> 'd2) + (avgX_R >> 'd2);
                  avgY_R2 <= avgY_R2 - (avgY_R2 >> 'd2) + (avgY_R >> 'd2);
                  sumX_R <= 30'b0;
                  sumY_R <= 30'b0;
                  cntr_R <= 19'b0;
            end
            //
*****************************************************************************************
************** //

            // ********** Spatial Filtering
************************************************************** //
            if (VGA_X1 == 10'b0) begin
                  data_reg2 [639:0] <= data_reg1 [639:0];
                  data_reg3 [639:0] <= data_reg2 [639:0];
                  data_reg4 [639:0] <= data_reg3 [639:0];
                  data_reg5 [639:0] <= data_reg4 [639:0];
                  data_reg6 [639:0] <= data_reg5 [639:0];
                  data_reg7 [639:0] <= data_reg6 [639:0];
                  data_reg8 [639:0] <= data_reg7 [639:0];
```

```
                data_reg9 [639:0] <= data_reg8 [639:0];
                data_reg10 [639:0] <= data_reg9 [639:0];

        end

        if ((data_reg2[VGA_X1-'d4]+data_reg2[VGA_X1-'d3]+data_reg2[VGA_X1-
'd2]+data_reg2[VGA_X1-'d1]+data_reg2[VGA_X1]
                +
data_reg2[VGA_X1+'d1]+data_reg2[VGA_X1+'d2]+data_reg2[VGA_X1+'d3]+data_reg2[VGA_X1+'d4
]
                + data_reg3[VGA_X1-'d4]+data_reg3[VGA_X1-'d3]+data_reg3[VGA_X1-
'd2]+data_reg3[VGA_X1-'d1]+data_reg3[VGA_X1]
                +
data_reg3[VGA_X1+'d1]+data_reg3[VGA_X1+'d2]+data_reg3[VGA_X1+'d3]+data_reg3[VGA_X1+'d4
]
                + data_reg4[VGA_X1-'d4]+data_reg4[VGA_X1-'d3]+data_reg4[VGA_X1-
'd2]+data_reg4[VGA_X1-'d1]+data_reg4[VGA_X1]
                +
data_reg4[VGA_X1+'d1]+data_reg4[VGA_X1+'d2]+data_reg4[VGA_X1+'d3]+data_reg4[VGA_X1+'d4
]
                + data_reg5[VGA_X1-'d4]+data_reg5[VGA_X1-'d3]+data_reg5[VGA_X1-
'd2]+data_reg5[VGA_X1-'d1]+data_reg5[VGA_X1]
                +
data_reg5[VGA_X1+'d1]+data_reg5[VGA_X1+'d2]+data_reg5[VGA_X1+'d3]+data_reg5[VGA_X1+'d4
]
                + data_reg6[VGA_X1-'d4]+data_reg6[VGA_X1-'d3]+data_reg6[VGA_X1-
'd2]+data_reg6[VGA_X1-'d1]+data_reg6[VGA_X1]
                +
data_reg6[VGA_X1+'d1]+data_reg6[VGA_X1+'d2]+data_reg6[VGA_X1+'d3]+data_reg6[VGA_X1+'d4
]
                + data_reg7[VGA_X1-'d4]+data_reg7[VGA_X1-'d3]+data_reg7[VGA_X1-
'd2]+data_reg7[VGA_X1-'d1]+data_reg7[VGA_X1]
                +
data_reg7[VGA_X1+'d1]+data_reg7[VGA_X1+'d2]+data_reg7[VGA_X1+'d3]+data_reg7[VGA_X1+'d4
]
                + data_reg8[VGA_X1-'d4]+data_reg8[VGA_X1-'d3]+data_reg8[VGA_X1-
'd2]+data_reg8[VGA_X1-'d1]+data_reg8[VGA_X1]
                +
data_reg8[VGA_X1+'d1]+data_reg8[VGA_X1+'d2]+data_reg8[VGA_X1+'d3]+data_reg8[VGA_X1+'d4
]
                + data_reg9[VGA_X1-'d4]+data_reg9[VGA_X1-'d3]+data_reg9[VGA_X1-
'd2]+data_reg9[VGA_X1-'d1]+data_reg9[VGA_X1]
                +
data_reg9[VGA_X1+'d1]+data_reg9[VGA_X1+'d2]+data_reg9[VGA_X1+'d3]+data_reg9[VGA_X1+'d4
]
                + data_reg10[VGA_X1-'d4]+data_reg10[VGA_X1-'d3]+data_reg10[VGA_X1-
'd2]+data_reg10[VGA_X1-'d1]+data_reg10[VGA_X1]
                +
data_reg10[VGA_X1+'d1]+data_reg10[VGA_X1+'d2]+data_reg10[VGA_X1+'d3]+data_reg10[VGA_X1
+'d4]) > 7'd75) begin
                fltr2 <= 10'h3FF;
                fltr2_R <= 10'h3FF;
                fltr2_G <= 10'h3FF;
                fltr2_B <= 10'h3FF;
        end

        else begin
                fltr2 <= 10'h0;
                fltr2_R <= 10'h0;
                fltr2_G <= 10'h0;
                fltr2_B <= 10'h0;
        end
```

```
              //
*********************************************************************************
************** //

              // ********** Thresholding
*************************************************************************** //
              if (((mRed-mGreen) > 10'd40) && ((mRed-mGreen) < 10'd296)) begin
                     //raw result
                     raw_R <= 10'h3FF;
                     raw_G <= 10'h3FF;
                     raw_B <= 10'h3FF;

                     data_reg1[VGA_X1] <= 1'b1; // * Update the bit according to the X
coordinate * //
              end

              else begin
                     //raw result
                     raw_R <= 10'h0;
                     raw_G <= 10'h0;
                     raw_B <= 10'h0;

                     data_reg1[VGA_X1] <= 1'b0; // * Update the bit according to the X
coordinate * //
              end
              //
*********************************************************************************
************** //

       end //else
end //always
```