Computer vision final project report

Ammar Nahari
Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, OH
axn337@case.edu

Mingxin Liu
Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, OH
mxl592@case.edu

I. Introduction:

Our goal for the project is to apply feature detection onto video images from a thermal camera pre-installed on the robot in the lab. Utilization and the application of thermal cameras are interesting and directly related to the field of computer vision. We first wrote launch file for the camera and node to run the camera in C++, then the rest of the feature detection code in Matlab. The project achieves a couple goals. First, the feature detection node can detect the location of a person and computes the centroid of the detected figure. Second, from that point on, the Matlab code contains a Gaussian filter algorithm that marks out the head of the detected figures and maximize their locations on the map. This approach can help sensors like thermal cameras to not only detect the approximate location, but also the exact points they are at on the map. Then, the robot can act according to the information acquired from the camera which compensates the lack of visibility at night for normal cameras.

II. Theory

$$H_{ij} = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{(i-(k+1))^2+(j-(k+1))^2}{2\sigma^2}\right); 1 \le i,j \le (2k+1$$

Figure 1.1

Figure 1.1 is the equation for a Gaussian filter kernel of size 2k by 2k, and figure 1.2 is how we can calculate the gradient values for Gx and Gy.

$$d(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2}$$

$$w(x,y) = \exp\left(-\frac{\sqrt{d(x,y)}}{2h^2}\right)$$

The kernel size for the convolution technique can directly affect the result of the filtered image. So the procedure of applying the canny edge detection algorithm is as follows. We first apply the Gaussian filter through convolution to smooth the image while removing the noises, then we compute the intensity gradient for the image. We then apply the threshold to determine the edges and their potential values, and we later track the edge for our centroid computation of interested area of a human body. In this case ts the head. The Gaussian filter is straightforward. We can set the filter kernel size as mentioned earlier 2k by 2k and apply convolution with the image matrix for Gaussian blurring. Then, we use the gradient equations ro detect horizontal, vertical and diagonal edges in the blurred image.

III. Centroid detection

```
<launch>
<node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    <param name="video_device" value="/dev/video2" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="framerate" value="30" />
    <param name="pixel_format" value="mjpeg" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap" />
</node>
<node name="to_mono_node1" pkg="image_proc" type="image_proc" ns="usb_cam" />
<node name="image_view" pkg="image_view" type="image_view" respawn="false" output="screen">
    <remap from="image" to="/usb_cam/image_mono" />
    <param name="autosize" value="true" />
</node></launch>
```

Here is the code for launching the thermal camera. The thermal camera was installed onto the robot with a positive 12V to the ground, powered by 4 Amperes serge current and comes down to 0.5 Amperes. The graphic information are accessed by ROS through openCV.

```
//edge detection
// publish result of processed image on topic "/image_converter/output_video"
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <std_msgs/Float64.h>
#include "kernels.h"

static const std::string OPENCV_WINDOW = "Open-CV display window";
using namespace std;

int g_brightratio; //threshold to decide if a pixel qualifies as dominantly "bright"
int edgeThresh = 1;
int const max lowThreshold = 100;
int ratio = 3;
int kernel size = 3;
char* window_name = "Edge Map";
Mat src, src_gray;
Mat dst, detected_edges;



class ImageConverter {
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    image_transport::Publisher image_pub_;
    ros::Publisher centroid_pub_;

public:

    ImageConverter(ros::NodeHandle &nodehandle)
    : it_(nh_) {
        // Subscribe to input video feed and publish output video feed
        image_sub_ = it_.subscribe("usb_cam/image_raw", 1,
                &ImageConverter::imageCb, this);
        image_pub_ = it_.advertise("/image_converter/output_video", 1);

        cv::namedWindow(OPENCV_WINDOW);
        centroid_pub_ = nodehandle.advertise<std_msgs::Float64>("/centroid", 1);
    }

    ~ImageConverter() {
        cv::destroyWindow(OPENCV_WINDOW);
    }

    //image comes in as a ROS message, but gets converted to an OpenCV type
    void imageCb(const sensor_msgs::ImageConstPtr& msg) {
        cv_bridge::CvImagePtr cv_ptr; //OpenCV data type
        try {
            cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::MONO16);//Changed RGB8 to MONO8 b
        } catch (cv_bridge::Exception& e) {
            ROS_ERROR("cv_bridge exception: %s", e.what());
            return;
        }
        // look for red pixels; turn all other pixels black, and turn red pixels white
        int npix = 0; //count the red pixels
        int isum = 0; //accumulate the column values of red pixels
        int jsum = 0; //accumulate the row values of red pixels
        int greyval;
        int gval;

    int testval;
        cv::Vec2b greypix; // OpenCV representation of an grey pixel
        //comb through all pixels (j,i)= (row,col)
        for (int i = 0; i < cv_ptr->image.cols; i++) {
            for (int j = 0; j < cv_ptr->image.rows; j++) {
                greypix = cv_ptr->image.at<cv::Vec2b>(j, i); //extract an RGB pixel
                //examine intensity of R, G and B components (0 to 255)
                greyval = greypix[0] ;
                gval = greypix[1];


                //look for red values that are large compared to blue+green

                //if red (enough), paint this white:
                if (greyval > 125) {
                        cv_ptr->image.at<cv::Vec3b>(j, i)[0] = 255;
                        cv_ptr->image.at<cv::Vec3b>(j, i)[1] = 255;
                        cv_ptr->image.at<cv::Vec3b>(j, i)[2] = 255;
                    npix++; //note that found another bright pixel
                    isum += i; //accumulate row and col index vals
                    jsum += j;
                    //ROS_WARN("Greyval is :%d and Gval is : %d ", jsum , isum );
                }
                else { //else paint it black
                        cv_ptr->image.at<cv::Vec3b>(j, i)[0] = 0;
                        cv_ptr->image.at<cv::Vec3b>(j, i)[1] = 0;
                        cv_ptr->image.at<cv::Vec3b>(j, i)[2] = 0;
                    }
            }
        }
```

```
        int half_box = 5; // choose size of box to paint
        int i_centroid, j_centroid;
        double x_centroid, y_centroid;
    //   if (npix > 0) {
            i_centroid = isum / npix; // average value of u component of red pixels
            j_centroid = jsum / npix; // avg v component
            x_centroid = ((double) isum)/((double) npix); //floating-pt version
            std_msgs::Float64 centroid_msg;
            centroid_msg.data = x_centroid;
            centroid_pub_.publish(centroid_msg);
            y_centroid = ((double) jsum)/((double) npix);
            ROS_INFO("u_avg: %f; v_avg: %f",x_centroid,y_centroid);
            //cout << "i_avg: " << i_centroid << endl; //i,j centroid of red pixels
            //cout << "j_avg: " << j_centroid << endl;
            for (int i_box = i_centroid - half_box; i_box <= i_centroid + half_box; i_box++) {
                for (int j_box = j_centroid - half_box; j_box <= j_centroid + half_box; j_box++) {
                    //make sure indices fit within the image
                    if ((i_box >= 0)&&(j_box >= 0)&&(i_box < cv_ptr->image.cols)&&(j_box < cv_ptr->image.rows))
                        cv_ptr->image.at<cv::Vec2b>(j_box, i_box)[0] = 255; //(255,0,0) is pure blue
                        cv_ptr->image.at<cv::Vec2b>(j_box, i_box)[1] = 0;
                        cv_ptr->image.at<cv::Vec2b>(j_box, i_box)[2] = 0;
                    }
                }
            }

    //   }
        // Update GUI Window; this will display processed images on the open-cv viewer.
        cv::imshow(OPENCV_WINDOW, cv_ptr->image);
        cv::waitKey(3); //need waitKey call to update OpenCV image window
        // Also, publish the processed image as a ROS message on a ROS topic
        // can view this stream in ROS with:
        //rosrun image_view image_view image:=/image_converter/output_video
        image_pub_.publish(cv_ptr->toImageMsg());

    }
    QImage canny(const QImage& input, float sigma, float tmin, float tmax) {//this could be a potential way to
        QImage res = convolution(gaussian_kernel(sigma), input), // Gaussian blur
    // Gradients
        gx = convolution(sobelx, res),
        gy = convolution(sobely, res);
        magnitude(res, gx, gy);
        // Non-maximum suppression
        quint8 *line;
        const quint8 *prev_line, *next_line, *gx_line, *gy_line;
        for (int y = 1; y < res.height() - 1; y++) {
            line = res.scanLine(y);
            prev_line = res.constScanLine(y - 1);
            next_line = res.constScanLine(y + 1);
            gx_line = gx.constScanLine(y);
            gy_line = gy.constScanLine(y);
        }
        for (int x = 1; x < res.width() - 1; x++) {
            double at = atan2(gy_line[x], gx_line[x]);
            const double dir = fmod(at + M_PI, M_PI) / M_PI * 8;
            if ((1 >= dir || dir > 7) && line[x - 1] < line[x] && line[x + 1] < line[x] ||
                (1 < dir || dir <= 3) && prev_line[x - 1] < line[x] && next_line[x + 1] < line[x] ||
                (3 < dir || dir <= 5) && prev_line[x] < line[x] && next_line[x] < line[x] ||
                (5 < dir || dir <= 7) && prev_line[x + 1] < line[x] && next_line[x - 1] < line[x])
            continue;
            line[x] = 0x00;
        }
    }
int main(int argc, char** argv) {
    ros::init(argc, argv, "bright_pixel_finder");
    ros::NodeHandle n; //
    ImageConverter ic(n); // instantiate object of class ImageConverter

    g_brightratio= 80; //choose a threshold to define what is "bright" enough
    ros::Duration timer(0.1);
    double x, y, z;
    while (ros::ok()) {
        ros::spinOnce();
        timer.sleep();
    }
    return 0;
}
```

The algorithm used for computing a centroid is straightforward. First, I used the ROS openCV library to create camera pointer that can access the image information and save them into a 2D matrix. Since the information of the camera is updating with respect to the refresh rate, the matrix I created was also a

float information with respect to the same refresh rate. The I set the threshold for the bright pixel (in this case is the brightness of the face) at 175 and filter out the lower pixels since they are inconsequential to our project. Thereafter, in a for loop of the length of the size of the matrix, I summed all the x rows and y rows to compute the centroid and display in the window.

IV. Morphological filtering

However, the centroid detection method only works for the case of a single person in the camera view. Therefore, in order to detect features for multiple people, we apply a different method.

The method we use here is image morphology, or in other words: erosion and dilation. We first store the CV converted image data into a matrix.

After that we convert the image matrix into grayscale first and binary filter it according to threshold. We determine the threshold value by setting it to the 80th percent of the maximum intensity pixel in the grayscale image. The binary filter will then discard all values under threshold and white paint the remaining pixels.

$$A \ominus B = \bigcap_{b \in B} A_{-b}$$

Then we apply a number of erosion iteration to the threshold matrix to filter out small white spaces in the binary image. This step will eliminate all noise and only keep the bright pixels that represent humans. In this case, according to the equation above, erosion is undergoing a translation of -b onto matrix A.

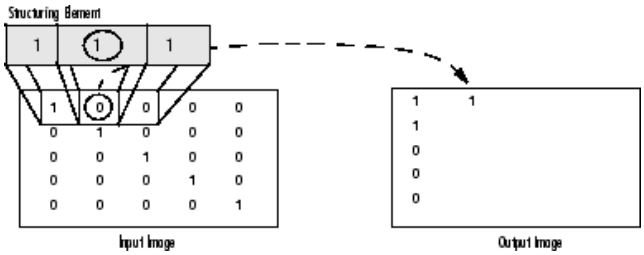


Figure 4.1

Then we apply dilation on the eroded image to increase all remaining white space. The step helps for better detection because it highlights the important features in the image and combine the white spaces for each individuals in the image. What the dilation operation doing is that it move a window of spicified shape and dimension over the image, in our application it is a square window of size 8 by 8. Then it checks the center element in the window, if it is true, it exaggerate the value of it and convert all the pixels under the window to true. However, the exaggeration is done one way, which means that if the element of entrest is false, it won't change the values of pixels under the window. Figure 4.1 retreived from (https://www.mathworks.com/help/images/morphological-dilation-and-erosion.html) explains the concept of dilation visually.

```
cv::Mat camera_feed;
cv::Mat threshold;
//max number of objects to be detected in frame
const int MAX_NUM_OBJECTS=50;
//minimum and maximum object area
const int MIN_OBJECT_AREA = 20*20;
const int MAX_OBJECT_AREA = FRAME_HEIGHT*FRAME_WIDTH/1.5;
//names that will appear at the top of each window

const string windowName3 = "After Morphological Operations";

void morphOps(cv::Mat &thresh){

    //create structuring element that will be used to "dilate" and "erode" image.
    //the element chosen here is a 3px by 3px rectangle

    cv::Mat erodeElement = cv::getStructuringElement( MORPH_RECT,Size(3,3));
    //dilate with larger element so make sure object is nicely visible
    cv::Mat dilateElement = cv::getStructuringElement( MORPH_RECT,Size(10,10));

    cv::erode(thresh,thresh,erodeElement);
    cv::erode(thresh,thresh,erodeElement);
    cv::erode(thresh,thresh,erodeElement);
    cv::erode(thresh,thresh,erodeElement);
    cv::erode(thresh,thresh,erodeElement);

    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);



}
```

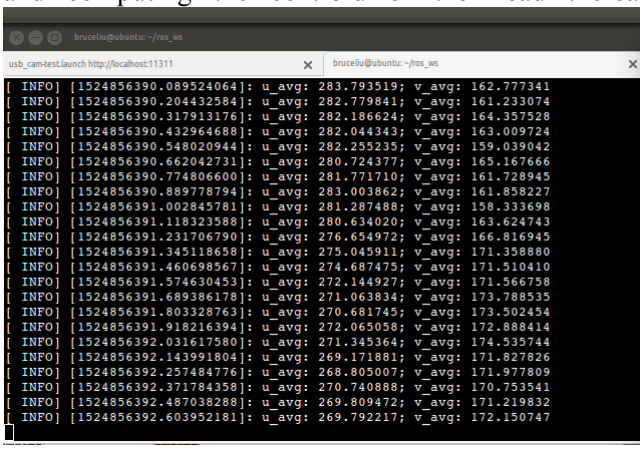
```
void trackFilteredObject(int &x, int &y, Mat threshold, Mat &cameraFeed){

    cv::Mat temp;
    threshold.copyTo(temp);
    //these two vectors needed for output of findContours
    vector< vector<Point> > contours;
    vector<Vec4i> hierarchy;
    //find contours of filtered image using openCV findContours function
    findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE );
    //use moments method to find our filtered object
    double refArea = 0;
    bool objectFound = false;
    if (hierarchy.size() > 0) {
        int numObjects = hierarchy.size();
        //if number of objects greater than MAX_NUM_OBJECTS we have a noisy filter
        if(numObjects<MAX_NUM_OBJECTS){
            for (int index = 0; index >= 0; index = hierarchy[index][0]) {

                Moments moment = moments((cv::Mat)contours[index]);
                double area = moment.m00;

                //if the area is less than 20 px by 20px then it is probably just noise
                //if the area is the same as the 3/2 of the image size, probably just a bad filter
                //we only want the object with the largest area so we safe a reference area each
                //iteration and compare it to the area in the next iteration.
                if(area>MIN_OBJECT_AREA && area<MAX_OBJECT_AREA && area>refArea){
                    x = moment.m10/area;
                    y = moment.m01/area;
                    objectFound = true;
                    refArea = area;
                }else objectFound = false;


            }
            //let user know you found an object
            if(objectFound ==true){
                putText(cameraFeed,"Tracking Object",Point(0,50),2,1,Scalar(0,255,0),2);
                //draw object location on screen
                drawObject(x,y,cameraFeed);}

        }else putText(cameraFeed,"TOO MUCH NOISE! ADJUST FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
    }
}
```

V. Results

Figure 2.1

Figure 2.2

Figure 1 is the result of the centroid computation. Compare with figure 2.2, figure 1 computed the centroid centered on the chest. This is because the centroid was calculated based on all the bright pixels in the window. But for our interest of the project, we are really trying to detect the head area of the person in order to distinguish between different individuals in the window. Figure 2.1 is the window that shows the computed centroid that doesn't contain any gaussian filter for the head. However, for figure 2.2, after applying the canny Gaussian filter, the centroid is successfully centered on the person's face and computing the centroid of the head thereafter.



```
[ INFO] [1524856390.089524064]: u_avg: 283.793519; v_avg: 162.777341
[ INFO] [1524856390.204432584]: u_avg: 282.779841; v_avg: 161.233074
[ INFO] [1524856390.317913176]: u_avg: 282.186624; v_avg: 164.357528
[ INFO] [1524856390.432964688]: u_avg: 282.044343; v_avg: 163.009724
[ INFO] [1524856390.548020944]: u_avg: 282.255235; v_avg: 159.039042
[ INFO] [1524856390.662042731]: u_avg: 280.724377; v_avg: 165.167666
[ INFO] [1524856390.774806600]: u_avg: 281.771710; v_avg: 161.728945
[ INFO] [1524856390.889778794]: u_avg: 283.003862; v_avg: 161.858227
[ INFO] [1524856391.002845781]: u_avg: 281.287488; v_avg: 158.333698
[ INFO] [1524856391.118323588]: u_avg: 280.634020; v_avg: 163.624743
[ INFO] [1524856391.231706790]: u_avg: 276.654972; v_avg: 166.816945
[ INFO] [1524856391.345118658]: u_avg: 275.045911; v_avg: 171.358880
[ INFO] [1524856391.460698567]: u_avg: 274.687475; v_avg: 171.510410
[ INFO] [1524856391.574630453]: u_avg: 272.144927; v_avg: 171.566758
[ INFO] [1524856391.689386178]: u_avg: 271.063834; v_avg: 173.788535
[ INFO] [1524856391.803328763]: u_avg: 270.681745; v_avg: 173.502454
[ INFO] [1524856391.918216394]: u_avg: 272.065058; v_avg: 172.888414
[ INFO] [1524856392.031617580]: u_avg: 271.345364; v_avg: 174.535744
[ INFO] [1524856392.143991804]: u_avg: 269.171881; v_avg: 171.827826
[ INFO] [1524856392.257484776]: u_avg: 268.805007; v_avg: 171.977809
[ INFO] [1524856392.371784358]: u_avg: 270.740888; v_avg: 170.753541
[ INFO] [1524856392.487038288]: u_avg: 269.809472; v_avg: 171.219832
[ INFO] [1524856392.603952181]: u_avg: 269.792217; v_avg: 172.150747
```

Figure 2.3

Figure 2.3 shows that the centroid is being computed and constantly updated while the while loop is still valid.
Figure 5 is the RGB image showing Ammar and Mingxin through the macbook usb camera. Figure 5.1 is showing the same figures but with the thermal camera on usb camera input 1. Then applying the erosion and dilation method, we have the following results. With erosion for 5 times, the program can successfully distinguish the features of the person within the thermal image and compute the head positions in the window view.

Figure 5

Figure 5.1

Figure 5.2

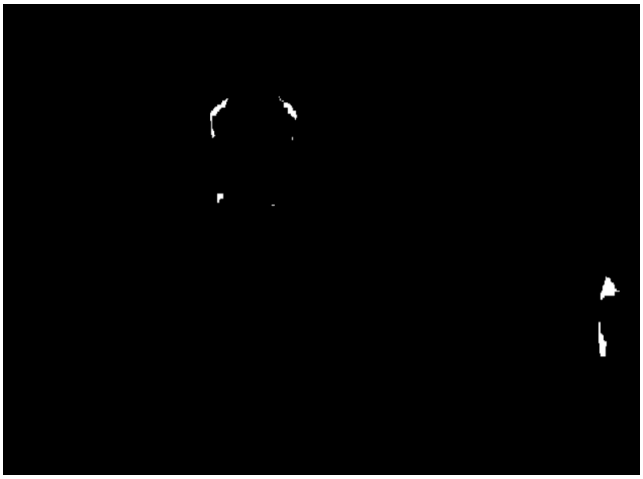


Figure 5.3

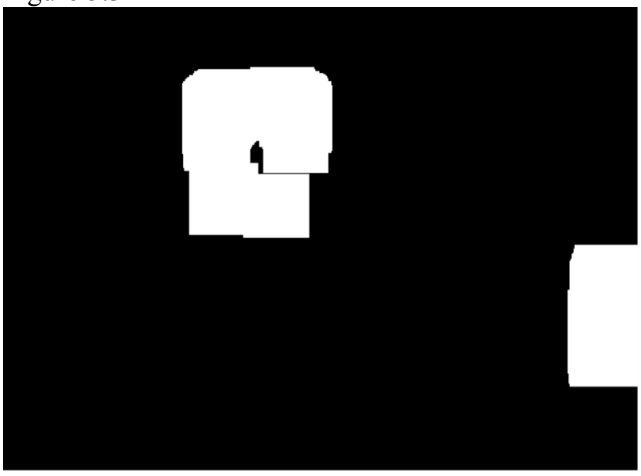


Figure 5.4

Figure 5.5

Figure 5.6

Figure 5.7

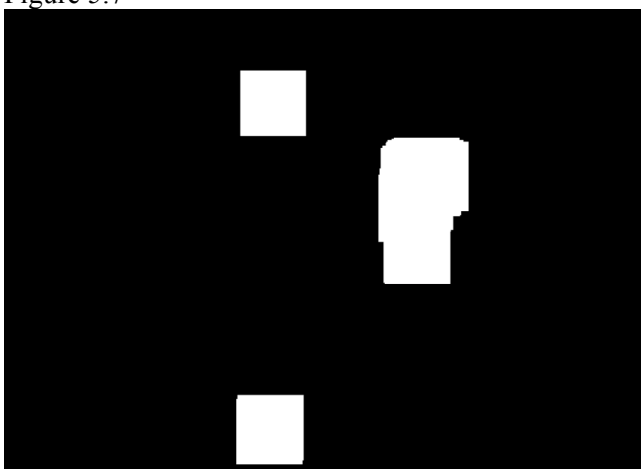


Figure 5.8

The second set of example shows when detecting features of multiple people in the figure. In figure 5.7, its relatively hard to see the bright dot of person on the left but there is a very small point among the dark space. However, there seems to be an error in the result showing in figure 5.8. There is also a bright pixel picked up by the program that was actually the hand of the person on the left. The reason this happened its due to the similar brightness of the hand with the face. However, increasing the threshold level may help reducing this error, however since the brightness it's too close, this method might result in erasing the light spot for the face as well. This is definitely an approach that we could take to improve on our future projects.

VI. Conclusion

With our methods of computer vision techniques, we were able to first compute the position of a person using the thermal camera. Second, we were able to detect the features of multiple people in the camera view and compute their positions. We use different methods to accomplish these results, such as Gaussian filtering, canny edge detection, bandwidth filtering and image morphology. Some of the challenges we faced is the compatibility of opencv2 in ROS, which was the compiling platform we used for the project, and latencies occurring because of computational expenses. However, we were able to overcome these problems by looking into alternative interfaces and more less time-consuming methods that suit our application for live tracking.

Nevertheless, there are some area we could improve on the future projects. First, we based most of our algorithms on intensity and color filtering. We would achieve better results if we include shape recognition algorithms into our project, and it would help us detecting different object from the thermal camera view in addition to humans. Secondly, we have been using ROS C++ interface for our project, and it is not the best platform for visual tracking application. Alternative platforms would be Visual Studios, Python CV, or Matlab Visual Tool box. Future improvements include: have a better result on multiple people recognition with a better feature detection algorithm; more accurate position finding with Gaussian filter; more active program that does updated feature checking.