HAND GESTURE RECOGNITION USING KINECT

Heng Du, TszHang To

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

December 15, 2011

Technical Report No. ECE-2011-04

Contents

1. **Introduction**………………………………………………………………….. 1

2. **Review of literature**…………………………………………………………….. 1

3. **Methods**…………………………………………………………………………….. 2

4. **Experimental results**………………………………………………………… 12

5. **Conclusions**………………………………………………………………………… 13

6. **References**………………………………………………………………………….. 13

7. **Appendix**…………………………………………………………………………… 15

List of Figures

Fig. 1 Overall workflow 2

Fig. 2 Workflow of hand extraction 3

Fig. 3 Original depth map 4

Fig. 4 Rescalin 4

Fig. 5 Background elimination 5

Fig. 6 Extraction 6

Fig. 7 Parameters optimization of the Shi-Tomasi corner detector 7

Fig. 8 Median filtering of the image 8

Fig. 9 Hand contour 9

Fig. 10 Hand contour with approximation polygon 9

Fig. 11 Hand contour with detected convexity points 10

Fig. 12 Hand contour with filtered convexity points 11

List of Tables

Table 1 Confusion matrix for performance evaluation 12

1. Introduction

The Kinect 3-D camera, with its depth sensing capability, has given birth to various exciting projects in human-machine interaction. One potential application is in the “wet lab” environment where a technician with gloves on can enter numbers into a computer without physically touching the computer, thus avoiding the procedure of removing and putting on gloves. In this project, a robust algorithm, which can recognize various arrangements of fingers of both hands showing digits from 0 to 5 using a Kinect, will be developed.

2. Review of literature

It has been proposed in the literature that covariance matching can serve as a robust and computationally feasible approach to action recognition.[1],[2] This approach, which involves computing the covariance matrices of feature vectors that represent an action, can potentially be useful in our 2-D hand gesture recognition problem as well. Nonetheless, this stochastic approach requires careful selection of features in order to achieve high classification rate.

In addition to covariance matching, another approach has been proposed in the literature by Jmaa and Mahdi.[3] Instead of building a dictionary of covariance matrices, this approach is based on analyzing three primary features extracted from an image: location of fingers, height of fingers, and the distance between each pair of fingers. A histogram is used to represent the detected fingers in order to extract the features for digit recognition. By considering the three geometric features, this approach is able to return the appropriate hand-digit without pre-computing a dictionary as in the case of covariance matching.

Although covariance matching is much more flexible in that it can be used in a variety of recognition tasks, the approach requires careful selection of feature vectors and its computational cost is potentially high. On the other hand, the solution Jmaa and Mahdi proposed is limited to only hand-digit recognition. However, the implementation of this approach is relatively easier and requires less processing. In addition to the Jmaa and Mahdi method, there are numerous applications that use polygon approximation to detect convexity points for hand-digit

recognition.[4],[6] The combined approach will be more accurate, especially in hand-digit recognition system.

3. Methods

Overall workflow

The overall workflow of our project is composed of 5 steps. The figure below illustrates the flowchart. The first step is to setup the programming environment, including OpenNI, OpenCV, Kinect and Visual C++. Step two (image capture) through step five (hand digit recognition) corresponds to the program we developed.

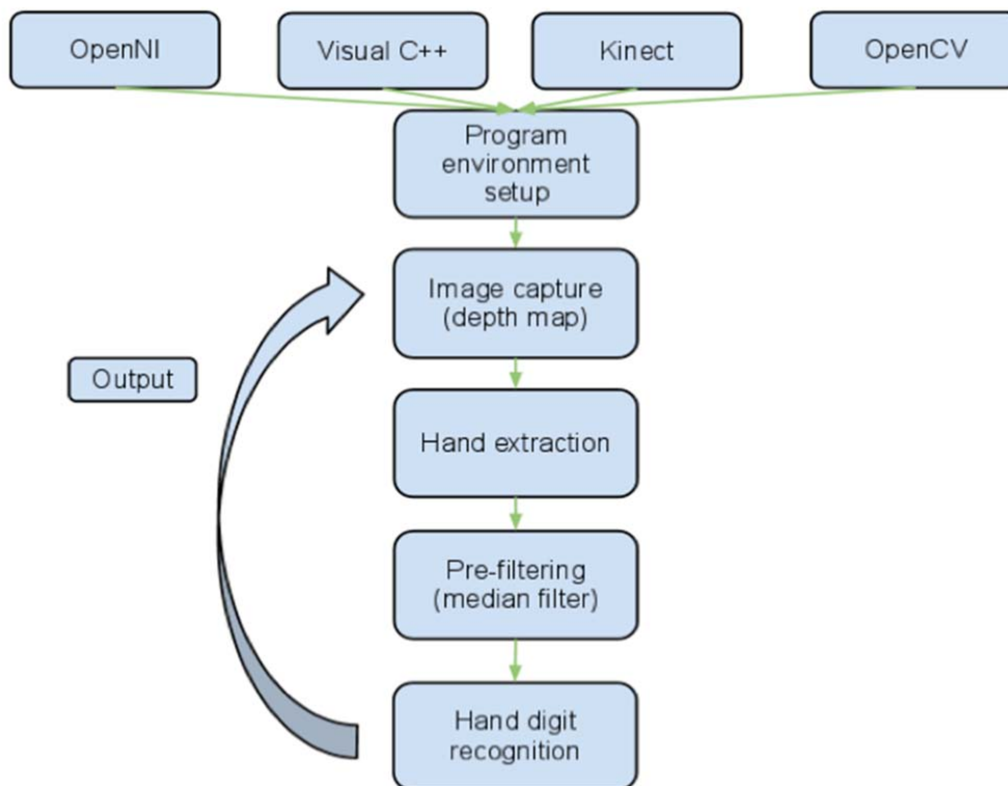

Figure 1 - Overall workflow

Image capture and extraction

Kinect can capture depth information by projecting an infrared dots pattern and its subsequent capture by an infrared camera. With this feature, we can easily get the shape of hand while discarding other information. The approach is described below.

The depth information that is captured will be converted into a gray scale image. The image does not contain any color information. When someone operates the Kinect, the person's hands should be in the front, so that the Kinect can extract the hands by judging the depth. All these are in real-time. The steps are described below.

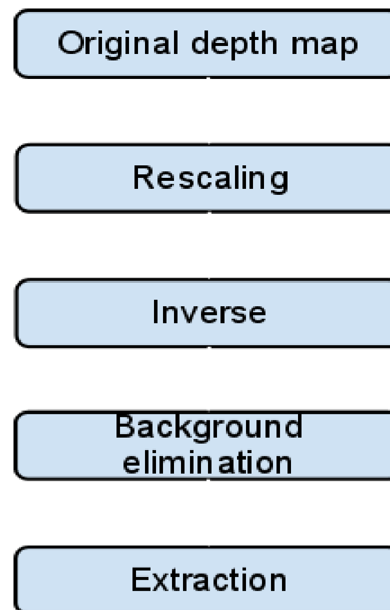

Figure 2 - Workflow of hand extraction

Original depth map

The first step is to obtain the image from the depth camera. With the OpenNI and OpenCV, we can obtain the images in 640*480 resolution at 30fps. Each frame is converted into a matrix in uint8 gray scale. However, the raw image is barely recognizable due to low color contrast, and thus the whole image is dark.

Figure 3 - Original depth map

Rescaling

To improve the raw image, we rescaled it. By adjusting the scale factor, we make the sensitive range to be from the minimum depth to about 1 meter from the Kinect. When the user is operating the Kinect correctly, this maximum depth should reach some point at the user's arm. The hand is in the sensitive range, so it's in gray scale, but the body and the background will be considered as the same, which is the maximum of depth, and thus not visible.

Figure 4 - Rescaling

Background elimination

The white background and the body are not necessary for extraction, so we performed a digital-negative operation on the image to make the image more optimal for subsequent steps. The shadows around the hand and the body, due to the positional disparity between the RGB camera and the depth sensor, can severely affect the performance of the recognition, so we have to eliminate them. Based on the previous step, the shadow at this point is white, while the background is black. The gray part is the hand, which is in the sensitive range. We can eliminate the shadows by thresholding the gray level.

Figure 5 - Background elimination

Extraction

To perform extraction, we have to improve our image further. We captured the top point of the palm of the hand, and kept the pixels which represent the depth up to 24 units from that point. Otherwise, they will be eliminated. All pixels in this range, will be converted into white (intensity thresholding). Finally we extracted a hand shape image for recognition.

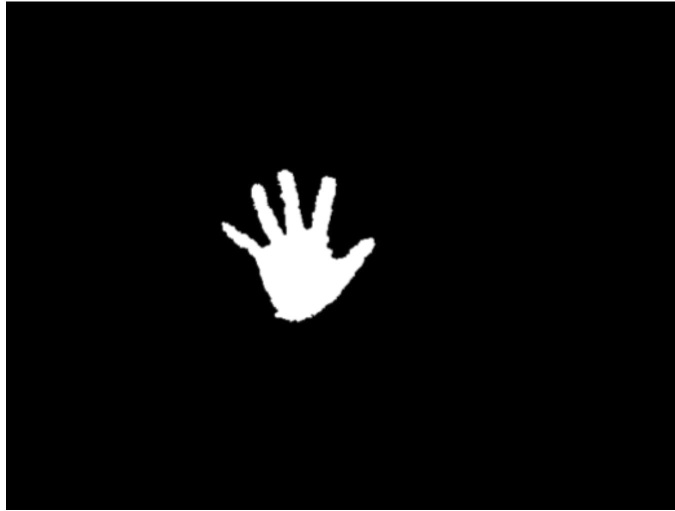

Figure 6 - Extraction

Corner detection

The original approach to hand-digit recognition is to use corner detection to build the relationship between corners and hand-digit gestures. Two common corner detection algorithms have been investigated: Harris corner detector and Shi-Tomasi corner detector. In Harris corner detector, the sum of squared differences (SSD) between a window in the image and its shifted-version is first computed:

$$S(x,y) = \sum_u \sum_v \xi(u,v)(f(u+x,v+y) - f(u,v))^2$$

which can be reduced to the following using Taylor expansion:

$$S(x,y) \approx [x\ y]\, A\, [x\ y]^T$$

Then the selection criteria score is computed by:

$$Score = detA - k(trace\ A)^2$$

In the above equation, A corresponds to the Harris matrix from the SSD, and k is the Harris detector free parameter. On the other hand, the selection score of the Shi-Tomasi

corner detector is computed by finding the minimum of the eigenvalues to the Harris matrix A:

$$Score = min(\lambda_1, \lambda_2);\ \ \lambda_1, \lambda_2\ are\ eigenvalues\ of\ A$$

In this project, Shi-Tomasi detector was chosen for testing. By varying the threshold score and the minimum distance between corners, the detector was able to return just a single corner for each raised finger:

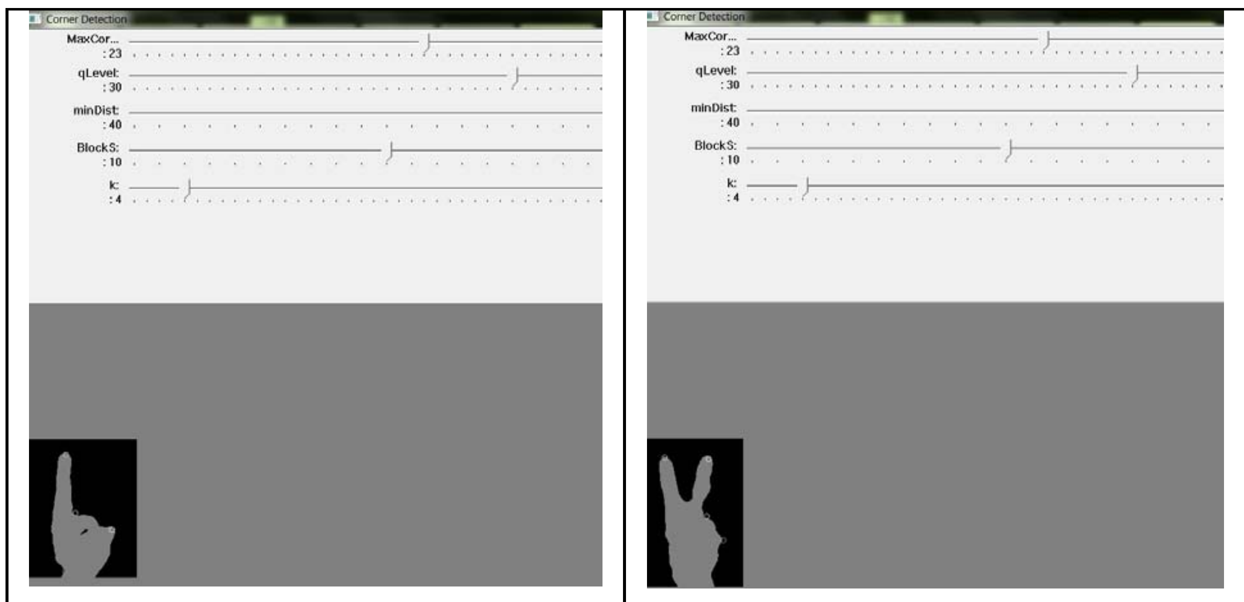

Figure 7 - Parameters optimization of the Shi-Tomasi corner detector

Although the Shi-Tomasi corner detector was able to return just one corner for each raised fingertip, there were multiple parameters that could potentially affect the output of the recognition. These parameters include the score threshold, the scaling factor of score, the block size of detector kernel, and the minimum distance between detected corners. Thus, we opted for another approach to the hand-digit recognition problem.

Convexity detection

The final approach to the hand-digit recognition is convexity detection. This approach attempts to locate the convex points and concave points of the hand shape. Then according to the number of convex and concave points detected, a digit is predicted. This approach was chosen over the original approach because it requires less number of parameters, and it proves to be more robust in our application.

The convexity detection approach consists of the following steps:

1. Pre-filtering of the image
   A median filter with a window size of 15 x 15 pixels was used to smooth the edges of the extracted image. This filtering operation can be written mathematically as:

$$g(i,j) = median(f(i+k, j+l)\, h(k,l))$$

where f is the source image, g is the filtered image, and h is the uniformly-weighted median-filtering kernel. This step was necessary in order to reduce image noise and the number of unwanted convexity points. The corresponding OpenCV function is <medianBlur>.

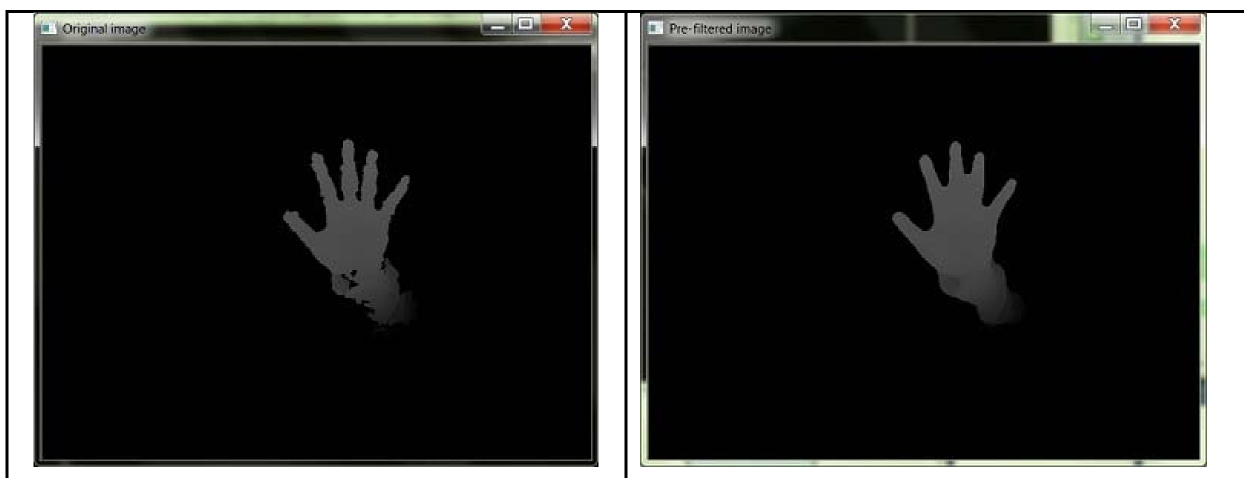

Figure 8 - Median filtering of the image

2. Tracing the hand contour

   To find the curvature points of the hand shape, the hand contour was traced using the OpenCV function <findContours>. This reduced the computational complexity in the subsequent steps by restricting the computations only to the hand contour but not the entire image.

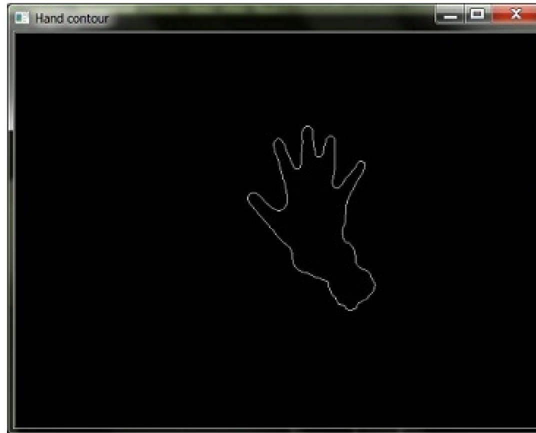

Figure 9 - Hand contour

3. Approximating the hand contour with a polygon

   This step further reduced the number of unwanted convexity points by approximating the hand contour with a polygon that has fewer vertices. The corresponding OpenCV function is <approxPolyDP>. This function is based on the Douglas-Peucker graph algorithm, which recursively connects a start point and an end point of a line segment by finding the vertex furthest away from the line segment.

Figure 10 - Hand contour with approximation polygon

4. Detection of convex and concave points of the approximation polygon

   This step was done using the OpenCV function <convexHull> and an adapted version of <cvConvexityDefects>. The convexHull function is based on the Sklansky's graph algorithm, which consists of the following steps[4]:

   a. label an external vertex p0, then label the rest of the vertices clock-wise
   b. place 3 coins on p0, p1, p2, and label them "back", "center", "front"
   c. iteratively re-label/remove vertexes until "front" is on p0 and 3 coins form a right turn

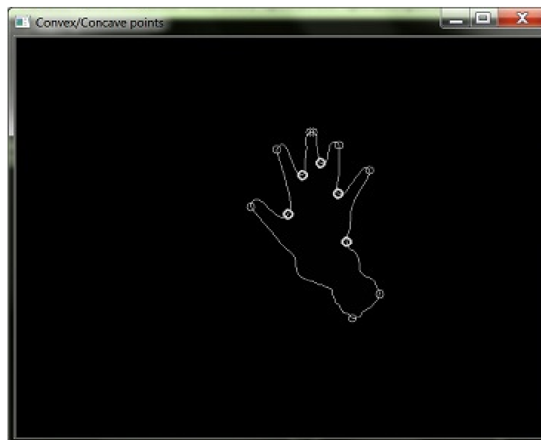

Figure 11 - Hand contour with detected convexity points

5. Filtering of convex and concave points

   The convexity points detected in the previous step were filtered in two steps:

   a. Grouping of neighboring points - for each convex or concave point, the distance between the current point and the next point was compared. If the distance was less than the distance threshold we defined, the current point and the next point were considered as a cluster, and the current point was filtered out.
   b. Filtering convex points not from finger tips - a minimum enclosing rectangle of the approximation polygon was computed. Then the center of palm was approximated by calculating the center point of the enclosing rectangle. Any convex point whose height (y-coordinate) was below the height (y-coordinate) of the center of the palm was filtered out.

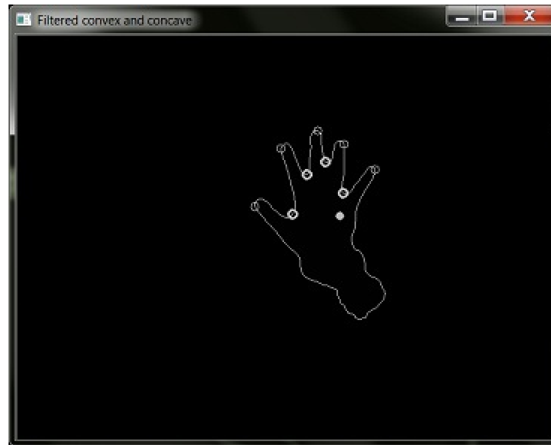

Figure 12 - Hand contour with filtered convexity points

6. Outputting the result based on the number of convex and concave points
   a. For digits 2 to 5, the predicted digit is simply the number of convex points, given there is at least one concave point.
   b. For digit 0, there is no concave point, and all the convex points need to be within a threshold radius from the approximated center of the palm.
   c. For digit 1, there is no concave point, and there is at least one convex point that is outside the threshold radius from the approximated center of the palm

4. Experimental results

To quantify the performance of the hand digit recognition algorithm in the aforementioned restrictions, a confusion matrix based on a set of 36 still images was computed. The testing image set consists of six images for each digit, and the six images represent two sets of conditions, with each set produced by a different person:

1. Gesture represented using the right hand
2. Gesture represented using the left hand
3. Gesture that is slightly rotated

The resulting confusion matrix is illustrated below:

		Predicted					
		0	1	2	3	4	5
Actual	0	5	1				
	1		6				
	2			6			
	3				5	1	
	4					6	
	5						6

Table 1 - Confusion matrix for performance evaluation

According to the table, the actual digits match correctly with the predicted digits, except for one false recognition for digit zero and one false recognition for digit three. Thus, for the testing images, the recognition algorithm has a correct classification rate of:

$$34 / 36 * 100\% = 94.4\%$$

5. Conclusions

In conclusion, the proposed recognition algorithm is able to detect hand digits from 0 to 5 with a correct classification rate of 94%. In addition, the algorithm can be implemented in real time in Visual Studio C++ environment with unnoticeable lag.

However, in order for the algorithm to work optimally, the following constraints have been imposed:

1. Only one hand at a time
2. The hand needs to be within a specific depth range (~70cm)
3. There should be no obstacle between the camera and the hand
4. The palm and forearm should be close to perpendicular
5. The fingers should be spread apart, and pointing upward

Given the restrictions imposed in order to achieve high classification rate, there are numerous potential improvements to the project as future work, including:

1. Ability to utilize two hands, from 0 to 10
2. Ability to extract hand shape flexibly (i.e. a wider depth range)
3. Ability to recognize digits from different orientations and rotations (i.e. the palm does not need to be at the front)
4. Minimizing real-time fluctuation in capturing image and outputting result
5. Ability to deploy the application for practical use

6. References

[1] - K. Guo, P. Ishwar, and J. Konrad. “Action Recognition in Video by Covariance Matching of Silhouette Tunnels”, Proc. of 22nd Brazilain Symposium on Computer Graphics and Image Processing, SIBGRAPI-2009,2009

[2] - O. Tuzel, F. Porikli, and P. Meer. “Region Covariance: A Fast Descriptor for Detection and Classification”. In Proc. ECCV (2) 2006, pp.589-600

[3] - A. B. Jmaa and W. Mahdi, “A New Approach For Digit Recognition Based On Hand Gesture Analysis”, International Journal of Computer Science and Information Security (IJCSIS), Vol 2, No 2, 2009

[4] - G.T. Toussaint. “The Three-Coins Algorithm for Convex Hulls of Polygons” <http://cgm.cs.mcgill.ca/~beezer/cs507/3coins.html>

[5] - OpenCV documentation - http://opencv.itseez.com/

[6] - L. Tongo - “A method of detecting and recognising hand gestures using OpenCV” http://www.andol.info/hci/1661.htm/

7. Appendix

C++ source code

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <opencv2\opencv.hpp>
#include <vector>
#include <cmath>
#include <opencv2/features2d/features2d.hpp>
#include <opencv\highgui.h>
#include <opencv2\imgproc\imgproc_c.h>
#include <stdarg.h>


using namespace cv;
using namespace std;

	////////////////////////////////////////////////////////////
	/// Global Variables
	int filterKernelSize = 15; // size of median filter, need to be odd number
	int resultDigit; // the outcome of the recognition algorithm


	Mat src; Mat srcSm;

	/// Function headers
	int getContourAndHull(cv::Mat);
	vector<int> elimNeighborHulls(vector<int>, vector<Point>); // to remove neighbor hulls
	vector<int> filterHulls(vector<int>, vector<Point>, RotatedRect); // to remove hulls below a height
	vector<int> filterHulls2(vector<int>, vector<Point>, vector<Point>, RotatedRect); // to further remove
hulls around palm
	vector<Point> filterDefects(vector<Point>, RotatedRect); // to remove defects below a height
	void findConvexityDefects(vector<Point>&, vector<int>&, vector<Point>&);
	void display(char*, cv::Mat);
	////////////////////////////////////////////////////////////
//	void cvShowManyImages(char* title, int nArgs, ...);


int main( int argc, char** argv )
{
	int i,j;
	int n;

	double handDepth=24;
	double maxVal1=0,minVal=0;
	unsigned char data=90;
	int lastNum=0;

	//Mat image;
	Mat num1,num2,num3,num4,num5,num0;
```

```
            const char* inFileSrc1= "C:/Users/HDR/Documents/Visual Studio 2010/Projects/openCV_tutorial/1.png";
            const char* inFileSrc2= "C:/Users/HDR/Documents/Visual Studio 2010/Projects/openCV_tutorial/2.png";
            const char* inFileSrc3= "C:/Users/HDR/Documents/Visual Studio 2010/Projects/openCV_tutorial/3.png";
            const char* inFileSrc4= "C:/Users/HDR/Documents/Visual Studio 2010/Projects/openCV_tutorial/4.png";
            const char* inFileSrc5= "C:/Users/HDR/Documents/Visual Studio 2010/Projects/openCV_tutorial/5.png";
            const char* inFileSrc0= "C:/Users/HDR/Documents/Visual Studio 2010/Projects/openCV_tutorial/0.png";
            num1 = imread( inFileSrc1, CV_LOAD_IMAGE_GRAYSCALE );
            num2 = imread( inFileSrc2, CV_LOAD_IMAGE_GRAYSCALE );
            num3 = imread( inFileSrc3, CV_LOAD_IMAGE_GRAYSCALE );
            num4 = imread( inFileSrc4, CV_LOAD_IMAGE_GRAYSCALE );
            num5 = imread( inFileSrc5, CV_LOAD_IMAGE_GRAYSCALE );
            num0 = imread( inFileSrc0, CV_LOAD_IMAGE_GRAYSCALE );



VideoCapture capture( CV_CAP_OPENNI );
if(!capture.isOpened())
   {
      cout << "Can not open a capture object." << endl;
      return -1;
   }
capture.set( CV_CAP_OPENNI_IMAGE_GENERATOR_OUTPUT_MODE, CV_CAP_OPENNI_VGA_30HZ );

cout << "\nDepth generator output mode:" << endl <<
      "FRAME_WIDTH    " << capture.get( CV_CAP_PROP_FRAME_WIDTH ) << endl <<
      "FRAME_HEIGHT   " << capture.get( CV_CAP_PROP_FRAME_HEIGHT ) << endl <<
      "FRAME_MAX_DEPTH    " << capture.get( CV_CAP_PROP_OPENNI_FRAME_MAX_DEPTH ) << " mm"
<< endl <<
      "FPS    " << capture.get( CV_CAP_PROP_FPS ) << endl;
for(;;)
{
            Mat image,image1;



            //dataptr=image.data;

   //capture.retrieve(image,CV_16UC1);
   if( !capture.grab() )
   {
      cout << "Can not grab images." << endl;
      return -1;
   }
   else
   {
      if(capture.retrieve(image, CV_CAP_OPENNI_DEPTH_MAP ) )
      {
//                        imshow("original", image);
                          //printf("image captured\n");
          const float scaleFactor = 0.3f;
                          image.convertTo( image, CV_8UC1, scaleFactor );
                          image=255-image;
                          //printf("image optimized\n");


                          n=image.cols;
```

```
uchar* dataptr=image.data;
//dataptr=image.data;
//printf("start shadow elimination\n");
for (i=0;i<=image.rows;i++)
{
        for (j=0;j<=image.cols;j++)
        {
                if(dataptr[n*i+j]==255)
                        dataptr[n*i+j]=0;
        };
};
image1=image.clone();
//printf("shadow elimination done\n");
//printf("start to calculate maxVal\n");
minMaxLoc(image,&minVal,&maxVal1,NULL,NULL);
//printf("the maxVal is %f\n",maxVal1);
//printf("the interger of maxVal is %u\n",(unsigned int)maxVal1);
//printf("the hand Depth is %u\n",handDepth);
uchar* dataptr1=image1.data;

for (i=0;i<image1.rows;i++)
{
        for (j=0;j<image1.cols;j++)
        {
                //data=dataptr1[n*i+j];
                /*if((unsigned int)dataptr1[n*i+j]<maxVal1-handDepth)
                        dataptr1[n*i+j]=0;
                else if(maxVal1!=255||maxVal1!=0)
                        dataptr1[n*i+j]=255;*/
                if(maxVal1<130&&maxVal1>24)
                        if((unsigned int)dataptr1[n*i+j]<maxVal1-handDepth)
                                dataptr1[n*i+j]=0;
                        else
                                dataptr1[n*i+j]=255;
                else
                        dataptr1[n*i+j]=0;


        };
};
//printf("processing finished\n");

imshow("depth map", image1);
cvMoveWindow("depth map",50,100);
//printf("image showed\n");
////////////////////////////////////////////////////////////////////////
medianBlur(image1, srcSm, filterKernelSize); // medianBlur smoothing filter

// processing to get result digit
 resultDigit = getContourAndHull(srcSm);
 if (resultDigit!=lastNum)
        if (resultDigit==0)
                {imshow("number",num0);
        cvMoveWindow("number",380,625);}
        else if (resultDigit==1)
                {imshow("number",num1);
```

```
                                        cvMoveWindow("number",380,625);}
                                        else if (resultDigit==2)
                                                {imshow("number",num2);
                                        cvMoveWindow("number",380,625);}
                                        else if (resultDigit==3)
                                                {imshow("number",num3);
                                        cvMoveWindow("number",380,625);}
                                        else if (resultDigit==4)
                                                {imshow("number",num4);
                                        cvMoveWindow("number",380,625);}
                                        else if (resultDigit==5)
                                           {imshow("number",num5);
                                        cvMoveWindow("number",380,625);}

                                 lastNum=resultDigit;
                                cout << "This is: " << resultDigit << endl;
                                ////////////////////////////////////////////////////////////////////////

        }

            }
//          capture>>image;
//          namedWindow( "kinect", CV_WINDOW_AUTOSIZE );
//          imshow( "kinect", image );
   if( waitKey( 30 ) >= 0 )
      break;
}
 waitKey(0);

 return 0;
}


/*
            adpated from
            http://opencv.itseez.com/doc/tutorials/imgproc/shapedescriptors/find_contours/find_contours.html#find-
contours
*/
int getContourAndHull(cv::Mat image) {
            // declarations
            Mat imageContour = image.clone(); // make a copy to work on
            vector< vector<Point> > contours; // all contours of image
            vector<Point> biggestContour; // the outermost contour
            vector<Point> approxContour; // to obtain polygon
            vector<int> hull; // convex points
            vector<int> filteredHulls; // filtered convex points
            vector<Point> defects; // concave points
            vector<Point> filteredDefects; // filtered concave points
            vector< Vec4i > hierarchy; // store informmation about contour
            double tmpContourArea1 = 0;
            double tmpContourArea2 = 0;
            double approxPolyDist = 15; // parameter to determine accuracy of appoximating polygon, the higher the
less accurate
            int r = 5; // radius of point for whatever being drawn
            RotatedRect minRect; // to roughly find the center of palm...
            Scalar color(rand()&255, rand()&255, rand()&255); // color of line when plotting (if it's on colored image)
```

```
	bool zeroOrOne = true; // initialized to zero = true, turn to one if false

	// obtain contour
	findContours(imageContour, contours, hierarchy,  CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE,
Point(0,0));
	// get the biggest (outmermost) contour
	vector< vector<Point> >::iterator iter;
	iter = contours.begin();
	while (iter != contours.end()){
		tmpContourArea1 = contourArea(*iter, false);
		if (tmpContourArea1 > tmpContourArea2) {
			tmpContourArea2 = tmpContourArea1;
			biggestContour = *iter;
		}
		iter++;
	}
	// quit method if no contour found
	if (biggestContour.empty() == true) {
		return -1;
	}
	/// draw contours
	drawContours(imageContour, contours, 0, color, 1, 8, hierarchy);

	/// get approximation polygon
	//double approxAcc = arcLength(biggestContour, false) / 40; // picked @ perimeter * 1/400
	approxPolyDP(biggestContour, approxContour, approxPolyDist, false); // somewhere between 5-30 is good
for apporixmation acc
	biggestContour = approxContour;

	/// find center of palm by the min enclosing rectangle
	minRect = minAreaRect(biggestContour);
	/// find convex and concave points
	convexHull(biggestContour, hull, false, false); /// get convex hull
	findConvexityDefects(biggestContour, hull, defects); /// get convexity defects

	/// filter convex/concave points
	filteredDefects = defects; // assign in case no filtering
	filteredDefects = filterDefects(defects, minRect);
	filteredHulls = hull;
	filteredHulls = filterHulls(hull, biggestContour, minRect);
	filteredHulls = elimNeighborHulls(filteredHulls,biggestContour);
	filteredHulls = filterHulls2(filteredHulls, filteredDefects, biggestContour, minRect);


	/// draw polygon
//	fillConvexPoly(imagePolygon, biggestContour, color, 8,0);
	/// draw enclosing rectangle and center
//	ellipse(imageContour, minRect,color,1,8);
	circle(imageContour, minRect.center, 2, color, 5, 8,0);
	for (unsigned int i=0; i < filteredDefects.size(); i++)
	{
		circle(imageContour, filteredDefects[i],r, color, 2, 8, 0);
	}
	for (unsigned int i=0; i<filteredHulls.size(); i++) {
		circle(imageContour, biggestContour[filteredHulls[i]], r, color, 1, 8, 0);
	}
```

```
	/// plot
	display("Filtered convex and concave points and center", imageContour);
	cvMoveWindow("Filtered convex and concave points and center",710,100);
//	cout << "Unfiltered defects: " << defects.size() << endl;
//	cout << "Unfiltered hull size: " << hull.size() << endl;
//	cout << "FilteredHull size: " << filteredHulls.size() << endl;
//	cout << "FilteredDefect size: " << filteredDefects.size() << endl;


	// determine resulting number of digits, if no convex defect found, use convex hull to determine if it's 0 or 1
	if (filteredDefects.size() > 0) {
		return filteredHulls.size(); // it's not 0 nor 1
	}
	else { // given no concave points detected, figure out if it's zero or one

		float palmRadius;
		if (minRect.size.height <= minRect.size.width) {
			palmRadius = (minRect.size.height)/2 ; // the normal case
		}
		else {
			palmRadius = (minRect.size.width)/2 ;
		}
		for (unsigned int i=0; i<filteredHulls.size(); i++ ) {
			if (biggestContour[filteredHulls[i]].y < (minRect.center.y - (palmRadius*2))) { //
multiply by two with trial and error
				zeroOrOne = false; // this is one
			}
		}
		if (zeroOrOne == true){
			return 0;
		}
		else if (zeroOrOne == false) {
			return 1;
		}
		else { // no digit assigned
			return -1;
		}

	}
}



/*	Function referenced from:
	http://stackoverflow.com/questions/6806637/convexity-defects-c-opencv
*/
void findConvexityDefects(vector<Point>& contour, vector<int>& hull, vector<Point>& convexDefects){
   if(hull.size() > 0 && contour.size() > 0){
   CvSeq* contourPoints;
   CvSeq* defects;
   CvMemStorage* storage;
   CvMemStorage* strDefects;
   CvMemStorage* contourStr;
   CvConvexityDefect *defectArray = 0;

   strDefects = cvCreateMemStorage();
```

```
   defects = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2, sizeof(CvSeq),sizeof(CvPoint), strDefects );

   // transform our vector<Point> into a CvSeq* object of CvPoint.
   contourStr = cvCreateMemStorage();
   contourPoints = cvCreateSeq(CV_SEQ_KIND_GENERIC|CV_32SC2, sizeof(CvSeq), sizeof(CvPoint),
contourStr);
   for(int i=0; i<(int)contour.size(); i++) {
      CvPoint cp = {contour[i].x,  contour[i].y};
      cvSeqPush(contourPoints, &cp);
   }

   // do the same thing with the hull index
   int count = (int)hull.size();
   int* hullK = (int*)malloc(count*sizeof(int));
   for(int i=0; i<count; i++){hullK[i] = hull.at(i);}
   CvMat hullMat = cvMat(1, count, CV_32SC1, hullK);

   // calculate convexity defects
   storage = cvCreateMemStorage(0);
   defects = cvConvexityDefects(contourPoints, &hullMat, storage);
   defectArray = (CvConvexityDefect*)malloc(sizeof(CvConvexityDefect)*defects->total);
   cvCvtSeqToArray(defects, defectArray, CV_WHOLE_SEQ);

   // store defects points in the convexDefects parameter.
   for(int i = 0; i<defects->total; i++){
      CvPoint ptf;
      ptf.x = defectArray[i].depth_point->x;
      ptf.y = defectArray[i].depth_point->y;
      convexDefects.push_back(ptf);
   }

   // release memory
   cvReleaseMemStorage(&contourStr);
   cvReleaseMemStorage(&strDefects);
   cvReleaseMemStorage(&storage);
   }
}

void display(char* window, cv::Mat image)
{
        namedWindow(window,CV_WINDOW_AUTOSIZE);
        imshow(window, image);
}

vector<int> elimNeighborHulls(vector<int> inputIndex, vector<Point> inputPoints) {
        vector<int> tempfilteredHulls;
        float distance;
        float distThreshold = 20;

        if (inputIndex.size() == 0) {
                return inputIndex; // it's empty
        }
        if (inputIndex.size() == 1) {
                return inputIndex; // only one hull
        }
        for (unsigned int i=0; i<inputIndex.size()-1 ; i++) { // eliminate points that are close
```

```
			distance = sqrt((float) pow((float) inputPoints[inputIndex[i]].x - inputPoints[inputIndex[i+1]].x, 2)
+ pow((float) inputPoints[inputIndex[i]].y - inputPoints[inputIndex[i+1]].y, 2));
			if ( distance > distThreshold ) { // set distance threshold to be 10
				tempfilteredHulls.push_back(inputIndex[i]);
			}
		}
		// get take of the last one, compare it with the first one
		distance  = sqrt((float) pow((float) inputPoints[inputIndex[0]].x - inputPoints[inputIndex[inputIndex.size()-
1]].x, 2) + pow((float) inputPoints[inputIndex[0]].y - inputPoints[inputIndex[inputIndex.size()-1]].y, 2));
		if ( distance > distThreshold ) { // set distance threshold to be 10
			tempfilteredHulls.push_back(inputIndex[inputIndex.size()-1]);
		}
		else if (inputIndex.size() == 2) { // the case when there are only two pts and they are together
			tempfilteredHulls.push_back(inputIndex[0]);
		}

		return tempfilteredHulls;
}

vector<int> filterHulls(vector<int> inputIndex, vector<Point> inputPoints, RotatedRect rect) {
		vector<int> tempFilteredHulls;
		float distThres = 20;
		for (unsigned int i=0; i < inputIndex.size(); i++) {
			if (inputPoints[inputIndex[i]].y < (rect.center.y + distThres)) { // 10 being threshold height
difference
				tempFilteredHulls.push_back(inputIndex[i]);
			}
		}
		return tempFilteredHulls;
}

vector<int> filterHulls2(vector<int> inputIndex, vector<Point> inputDefects, vector<Point> inputPoints,
RotatedRect rect) {
		if (inputIndex.size() > 2 && inputDefects.size() > 1) {
			return inputIndex;
		}
		// only do filtering if there are less than 3 convex points
		vector<int> tempFilteredHulls;
		float palmRadius;
		if (rect.size.height <= rect.size.width) {
			palmRadius = (rect.size.height)/2 ; // the normal case
		}
		else {
			palmRadius = (rect.size.width)/2 ;
		}
		// for now ignore angle or rotation
		for (unsigned int i=0; i < inputIndex.size(); i++) {
			if (inputPoints[inputIndex[i]].y < (rect.center.y - palmRadius)) { // 10 being threshold height
difference
				tempFilteredHulls.push_back(inputIndex[i]);
			}
		}
		return tempFilteredHulls;
}
```

```
vector<Point> filterDefects(vector<Point> inputDefects, RotatedRect rect) {
	vector<Point> tempFilteredDefects;
	for (unsigned int i=0; i <inputDefects.size(); i++) {
		if (inputDefects[i].y < (rect.center.y + 10)) {
			tempFilteredDefects.push_back(inputDefects[i]);
		}
	}
	return tempFilteredDefects;
}
```