University of Edinburgh
School of Informatics

AV Assignment 3

Andrew Burnie
Murray Settle

April 23, 2012

Abstract: This report details the work done and algorithms used in the creation of a Matlab program that would manipulate data taken from a Kinect Sensor as instructed inthe third assignment for the Advanced Vision course at Edinburgh University

Contents

1. Introduction

We have been given the task of processing data taken from a Kinect [1] sensor and using this to make changes to the video. In the 36 frame video a man is shown walking past a wall holding a black leaverarch folder in his swinging arm as he walks. This video is to be adapted firstly to change the bakground the man walks across to an image of a field of poppies. This given image would be placed within the section of wall shown in red in figure 1.1. The second adaptation is to place a video, of our choice, within the bounds of the folder the man is carrying in each frame so that video plays as he walks. For that reason we have chosen a video called Dramatic Chipmunk, frames in figure 1.2.

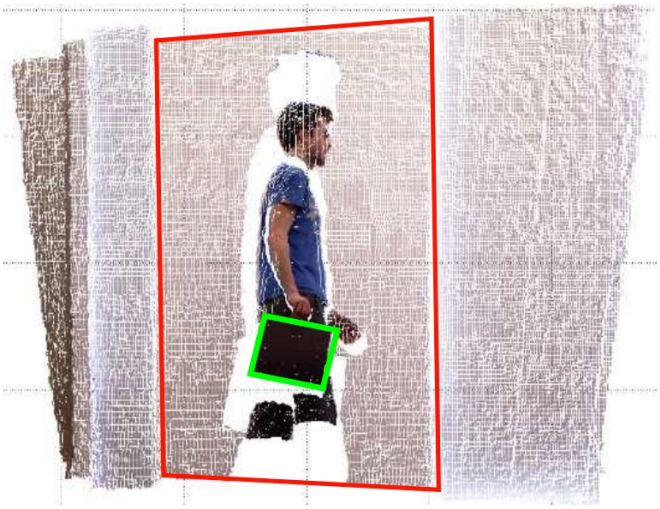


Figure 1.1: Section of wall highlighted in red

Figure 1.2: Dramatic Chipmunk frames

1.1 Data Structure Modification

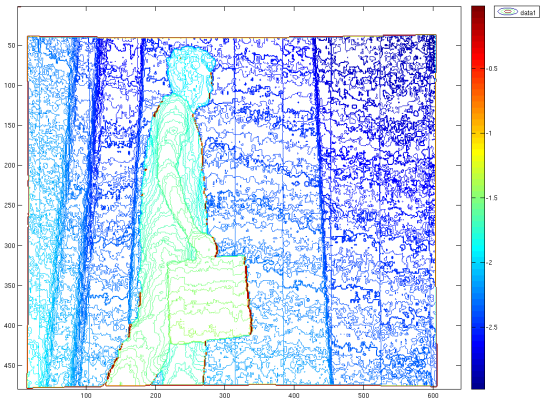
1.1.1 Image Data

The given data was in two parts, the first was a set of 36 images with width 640 pixels and height 480 pixels. These make up the frames of the original video split into red, green and blue components.

1.1.2 Depth Data

The second section of the data is the depth points. These correspond to each pixel's location in space giving them an x, y and z co-ordinate. The z component describes the depth into the image, the x the position from left to right and the y component the position from top to bottom.

On closer analysis it was seen that the z component of this data was stored effectively in countour lines as can be seen in figure 1.1.2. In this image we can see the boundary lines between the planes in which each pixel's z value is identical. It can be seen from the colouring that the contour lines that make up the vast majority of the image have values between -1.25 and -3. However there are also certain points with value 0, these pixels seem to be those where the Kinect Sensor cannot get a depth reading or is confused by steep edges in the depth data for example on the right hand endge of the folder. These artefacts in the data are not present around entire edges so are not useful for detecting the edges of objects. THey appear mostly to occur on pixels that are a part of the background so when we are separating background and foreground we should ensure they are included in the background as a rule.



The distance between the contours in our background data is approximately 0.0013, this means that we have an expected noise level of half this value as the values on boundaries between contours will be that distance from the real value. This noise level would be accurate if the kinect sensor was 100% accurate which no sensor is so in reality the expected noise level will be higher than this value and amplified by the fact that an incorrect reading must be at least 0.0013 off.

2. Overlaying the Field Image

We overlaid the provided image of the field onto the panel at the back of the image outlined in red in figure 1.1, as instructed by the assignment, using a homographic image transfer.

Code was provided to help write this part of the program in esthomog.m and remap.m from the IVR website. The latter was modified and named remapField.m, which is a function that begins by importing the image of the field. It then sets the target points to be the 4 corners of the panel (which were predetermined) and sets the source points to be the 4 corners of the image of the field. These points are then passed in to the esthomog function, which estimates the transformation matrix which will be used to map the points from the field image to where they should be on the background image. Finally it uses that matrix on each point in the background image to find if any pixel from the field image is to be transferred onto it.

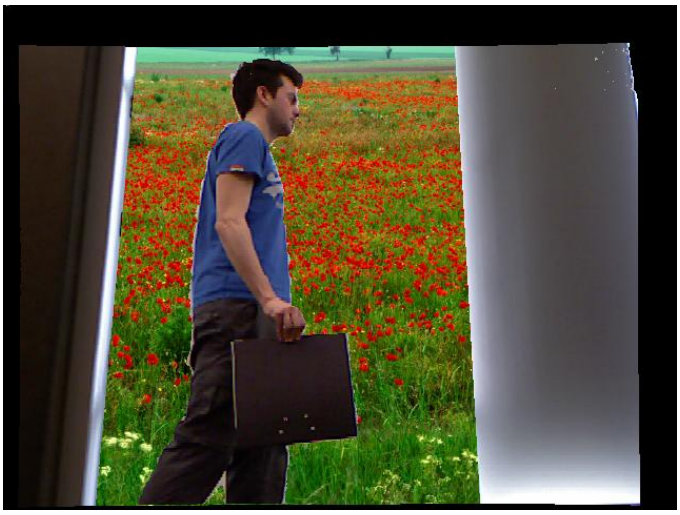
3. Extracting the Man

The first task was to work out which of the pixels represented the man in the image. As the video plays the lighting seems to change greatly, this is likely to be due to the camera changing settings, such as aperture size, automatically. This change in lighting would make it difficult to use colours to separate the man from the background ss we do not have consistend background colours. For that reason we chose to use the depth data.

The background is not flat as we can see from figure 1.1.2. There 4 separate sections of wall with different depths. Given the kinect sensor and wall are stationary we decided that it would simplify finding the man to use the depth points for frame one as a standard background. Firstly the z values were normalised to have values between 0 and 1 in each of the images. We then subtracted the initial background from each of the following images.

We subtracted the z co-ordinates of this initial image from all of the z-co-ordinates of each of the successive frames. This allowed us to set a threshold of minimum 0.03 to separate out the background. However this did not deal with the artefacts that existed around the man as these had been assigned a value of 1 when we normalised the data being the points recorded as furthest forwards. For that reason we needed an upper bound as well and found that a maximum threshold of 0.45 was able to isolate just the man in the image with no artefacts. However we also found that in some images small holes would appear in the head. Most of these could be removed using the supplied cleanup function however some still remained in some images.

The thresholds were used to create a binary image that represented the pixels we believed were just the man as in figure 3 taken from frame 18 of the original video. We could then overlay the man onto any image by taking the binary image for each frame and the full colour image and changing the pixels that were 1s in the binary image to have the same values as the pixels of the frame from which we have obtained the man. This overlaying can be placed over the new background created in the previous section for each frame giving us a video of the man walking past the field of poppies as required. The results of this are shown in figure 3.



4. Extracting the Black Quadrilateral

We set the task of finding the black quadrilateral in the image as creating a binary image where ideally all of the pixels in the quadrilateral were white and all other pixels black. This was not an easy task, and in the end the most effective method we found was one that was not a very general, adaptable solution.

4.1 Hard Thresholding Using the Depth

The first method we used to find the quadrilateral was to take the depth data and firstly normalise it so that the background pixels would be 0 and the closest part of the person would have a depth value of 1. Value for which there was no data in the original image were also set to 0. Then we took a binary image where the pixels of value 1 were the ones whose normalised depth value was above a hard threshold and the rest had value 0. Through trial and error with various images, the best fitting threshold on the given data to 3 decimal places was found for creating this binary image.

At this point the code also finds the value of the 13000th highest depth value in the image (as the quadrilateral was deemed to be the 13000 closest pixels in the image). However, since taking the closest 13000 pixels would not work for images where the quadrilateral was only partially in view or not at all, the program takes the previously computed binary image, and if the area of white pixels is not too small or too close to the side of the image then the program sets the 13000 pixels with the highest normalised depth values to be the white pixels in the new binary image, otherwise it sets the new binary image to be a completely black image. This new binary image is then the one that is passed on to the boundary tracking code in the best performing version of the program. This code was only meant to be part of a first attempt at creating this program, however, and we were not satisfied with the performance or sophistication of the method so we decided to try something different: fitting a plane to the quadrilateral, as discussed in the following section.

4.2 Plane Fitting Method

To try to improve performance we decided to look into finding the black quadrilateral by searching for flat planes in the image. A flat plane can be defined by an equation of the form:

$Ax + By + Cz + D = 0$

To find flat parts we scanned the image looking at successive 50 pixel by 50 pixel square sections. If these contained only foreground pixels as found in the binary images we had created earlier. For each section we had to decide if it described a flat part of the image. To do this we wanted to create a model plane that fitted the surface as well as possible. If that model plane fitted the points in the surface then we would know that that section was a flat plane.

4.2.1 Fitting A Model Surface

Knowing that our data was split into contours and that there was noise in the data we decided to use robust regression to create the model plane. We chose to use RanSaC [3] as it's algorithm allows for a fit to be found to the inlier data ignoring outliers. In our case the algorithm [2] does the following:

1. selects N pixel locations at random
2. estimates plane that best fits these data points
3. finds how many other pixels fit the model within a user given tolerance. Call this K.
4. if K is big enough, accept fit and exit with success
5. repeat the above L times
6. if you get here return the best fit found

We wish to minimise the chances that our algorithm will fail which we can do by setting the parameters ideally. This can be done with the following equation. Where p_{fail} is the probability of L consecutive failures and p_g is the probability that a pixel is a good fit to the model.

$L = \frac{log(p_{fail})}{log(1-(p_g)^N)}$

From this we can see that, assuming $p_g < 1$, the larger N is the smaller the denominator will be in the equation and hence the more runs we will need. Hence we wish to minimise N. To fit a flat 3D plane we need a minimum of 3 points so we shall set N to be 3. We wish to minimise the probability of failure, so set pfail to be 0.001, giving us an expectation of 1 failure in 1000 runs.

We can approximate pg by looking at the images we have. Allowing a tolerance equal to half of the distance between contours we can then estimate the number of wrongly classified pixels from the flat plane in the depth data. To do this we took one image and created lines of best fit through each contour line. In each section in between we then calculated the number of pixels that were incorrectly classified assuming our line of best fit was the correct contour. This gave us an approximate p_g of 0.87 as 87% of the points were classified correctly.

From this we can calculate L, using the above equation, as 6.49 which rounds up to 7 runs to reduce the probability of failure to 0.001.

4.2.2 Fitting a Plane To Our Points

Once we have chosen our points we wish to fit a plane to them, to do this we want to create vectors between the points in the plane and use them to calculate a normal to the plane. From this we can calculate the parameters of the plane thus:

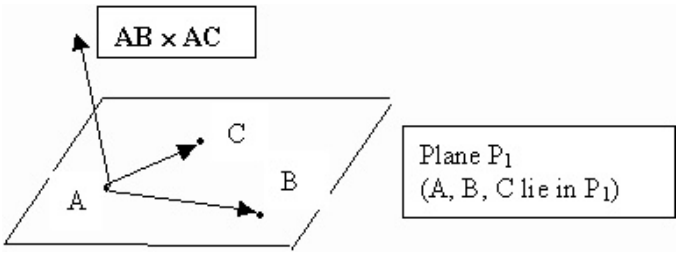


Figure 4.1: Calculating the normal of a plane

The normal is calculated by working out the cross product (cp_x, cp_y, cp_z) of two of the vectors in the plane. By then taking any of the points (a_x, a_y, a_z) in the plane we can then calculate the equation of the plane thus:

$cp_x * (x - a_x) + cp_y * (y - a_y) + cp_z * (z - a_z) = 0$

4.2.3 Checking The Fit Of Our Model Plane

To decide how close a fit our plane is we checked every point within the section we had selected and compared it to the model plane we had created. At this point we allowed for some noise by setting a tolerance for points. This tolerance was set to 0.01 to allow for the noise caused both by the distance between contours and wrongly read distance measurements. We can then calculate the percentage of points that fit the plane within this tolerance and use this as a measure of the quality of our fit.

4.2.4 Deciding If A Fit Is Good Enough

Ideally we would like all of the points in the section we are fitting to be withing the given tolerance of the plane. However given the noise in the data we were willing to accept that the area we were examining described a plane if 99% of the points fitted within our tolerance.

4.2.5 Finding The Quadrelateral

Once we had found a planar area we then had to expand this to try and fit the whole quadrelateral. To do that we searched for pixels that were within the foreground binary and whose pixels fitted the plane we had found. To do this we simply selected the forground pixels whose depth points also fitted the plane we had found. We then used the given cleanup function to erode and dilate the binary image into a cleaner binary and then used get largest to ensure we only labelled one area as the quadrelateral.

5. Overlaying the Black Quadrilateral

5.1 Boundary Tracking

Once we had this binary image of the isolated quadrilateral, we needed to find its 4 corners to be used for the homographic transfer. However, before we could find the corners, we needed to find the outline of the quadrilateral - at least for a sophisticated approach to finding the corners. The algorithm we wrote was based heavily on the corner finding algorithm used in the AV (Advanced Vision) lecture slides [4].

Firstly, the algorithm obtains the list of points that lie on the perimiter of the binary image using the bwperim function available in Matlab's Image Processing Toolbox [5]. This set of points is still not of use for finding the corners, as the algorithm for finding the corners requires the points to be ordered. For the points to be ordered, they have to be put through a function that tracks the boundary.

To track the boundary, we used a function called newBoundaryTrack which was based on the Matlab file boundarytrack.m from the AV website. Like the code it was based on, the function starts from the leftmost point in the image, then selects a pixel in the chosen pixel's 8 neighbouring pixels that both has not already been selected and is in the list of pixels originally passed to the boundary tracker. It then continues to select pixels this way until the conditions for a neighbouring pixel to select are not met.

There was, however, a problem with this algorithm. If the perimiter obtained by bwperim has dangling pixels (as in figure 5.1) then the boundary tracker will follow these and then terminate as there are no unvisited pixels from the original perimiter within the current pixel's neighbours.

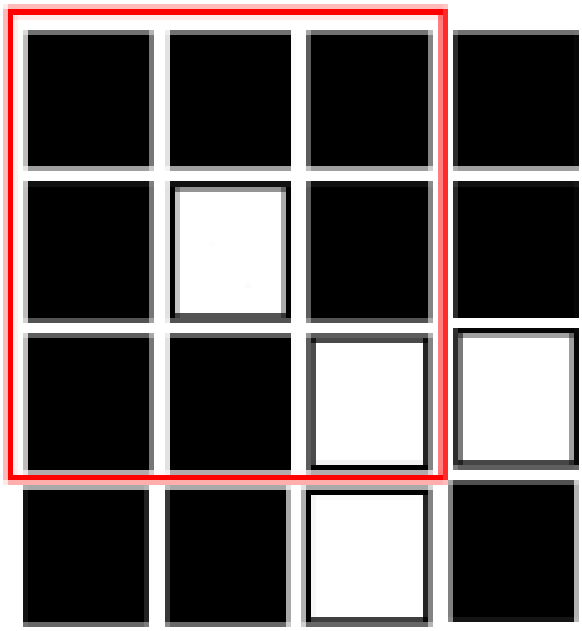


Figure 5.1: The boundary tracker would go down the route of the dangling pixel and terminate prematurely

So these dangling pixels had to be removed, and to do this we used a file from the AV website called removespurs.m. This worked well in removing the previously described dangling pixels, and this additionally removed unnecessary pixels along diagonal lines in the boundary, as seen in figure 5.2 The pixels in the line are labelled with '*', the unnecessary pixel with a 'c', pixels that are not taken into account with a '?' and black pixels are shaded in.

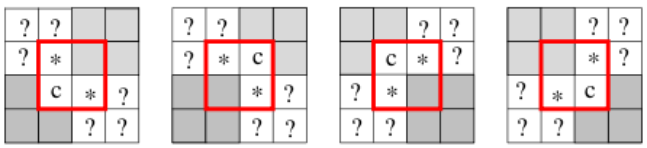


Figure 5.2: Situations in which unnecessary pixels are removed

However, it was in removing unecessary pixels that this function caused problems. The function only looked at square neighbourhoods of 4 pixels (seen outlined in red in figure 5.2) instead of neighbourhoods of 16 that should have been considered for removing these pixels, and so it would remove pixels that were needed to form the boundary. Larger neighbourhoods should have been considered for the following case which was observed when processing the given input data.

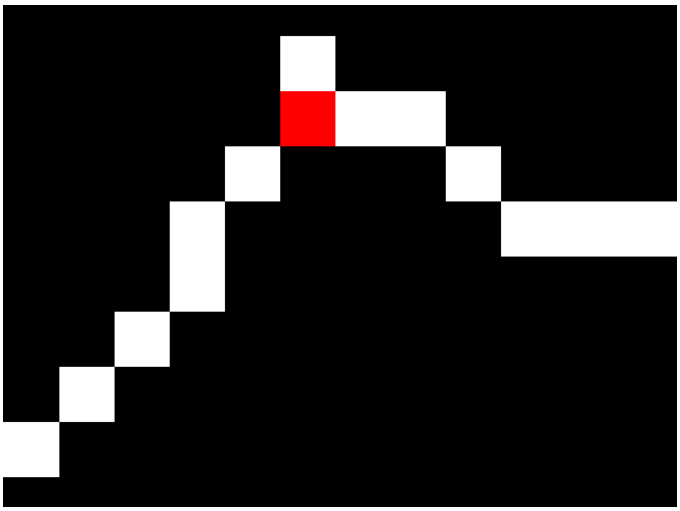


Figure 5.3: In this case the pixel highlighted in red would be wrongly removed using removespurs.m, causing the boundary tracker to fail

To account for this case, newRemoveSpurs was written as a slight modification of removespurs which would consider all of the pixels in the pictures in figure 5.2, not just the pixels outlined in red.

There was one other problem that we encountered during the boundary tracking process. This problem was with the code for removing dangling pixels and stray lines that would take the boundary tracker off course. The newRemoveSpurs function removes stray lines by repeatedly removing any pixels that have a maximum of 1 pixel out of its 8 neighbours being in the list of boundary pixels, but if there are stray lines with anything with more than a thickness of 1 pixel at the end, as in figure 5.4 these lines will not be removed and will throw off the boundary tracker.

For this reason, newBoundaryTrack was written as a modification of boundarytrack that would solve this problem. We changed it so that when the tracker terminated, it would remember the last pixel it passed that it had also passed earlier on, or the pixel when its last loop was created. It would then cut off any pixels from before the loop started. The quardrilateral formed a loop, as did these problem causing stray lines that were thicker at the top, so to avoid the problem of the boundary tracker getting stuck, the algorithm simply counts the number of pixels received from newBoundaryTrack and as the quadrilateral has many more pixels than the noisy loops at the top of the stray lines, this could be used to decide if the boundary found was that of the quadrilateral or not. If it was, the algorithm continued on to the corner finding algorithm, if not, the pixels found were removed from the original boundary and the newRemoveSpurs and newBoundaryTrack functions were applied again until the quadrilateral was

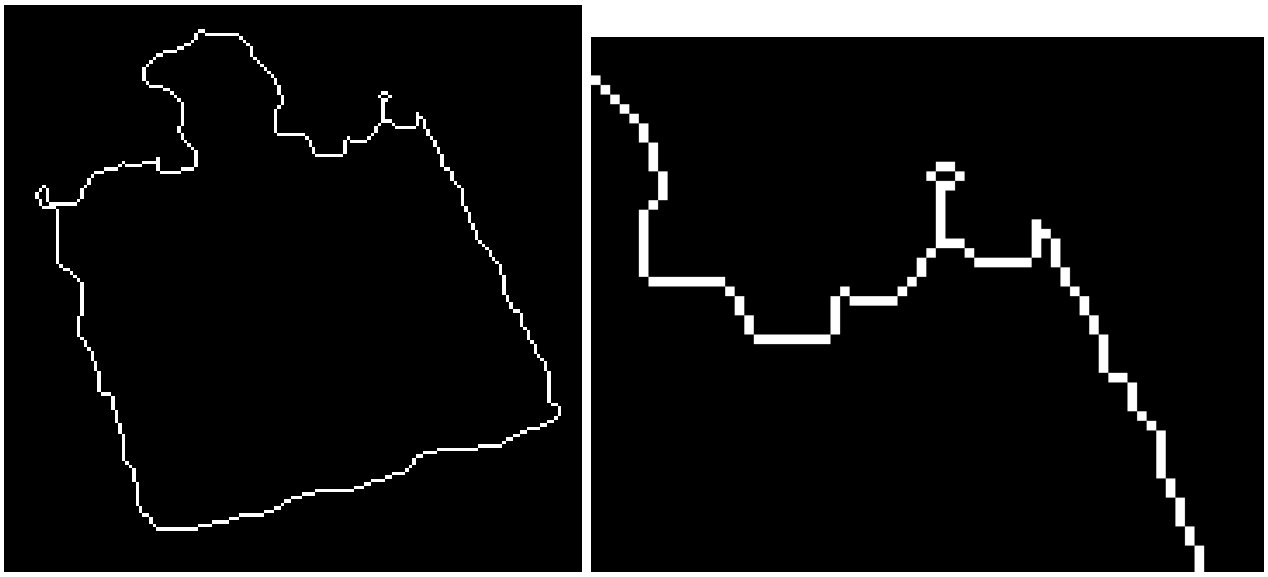


Figure 5.4: When the thickness at the end of the stray line is more than 1 this will cause the boundary tracker to fail

found.

5.2 Corner Finding

Once the boundary tracking process is complete, the corners can be found. The corner finding algorithm begins with a modified version of the algorithm in findcorners.m from the AV website (called newFindCorners.m). The algorithm is as follows:

1. Find leftmost point A
2. Find rightmost point B
3. Split points into the line from A to B and the line from B to A (using the ordering obtained by boundary tracking)
4. For each line
   (a) Find straight line between endpoints X and Y
   (b) Find point Z in the set of points which is furthest from the straight line by distance d (in pixels)
   (c) If d is less than a threshold, add this line

   (d) Else, split the set of points into the points between X and Z and the points from Z to Y and recursively perform this line splitting algorithm on those new lines

5. For each line, calculate and store its gradient
6. For each set of 2 lines that share an endpoint
   (a) Decide if gradients of lines are similar
   (b) If they are, merge them by replacing both lines with a line from the endpoints the lines did not have in common, recalculate the gradient for this line, and go back to 6, and recurse through all lines again

While the code regularly found the bottom 2 corners of the quadrilateral well, it was hard to find the top 2 corners as many corners were found around the top of the binary image by the corner finding algorithm (as in the pictures in figure 5.5), and even if the correct top 2 corners were selected, these were often lower than they should be as the binary images that were input into the algorithm sometimes did not fully extend to the top corners of the quadrilateral in the original image. So we wrote seperate code to subsequently take the bottom corner and assume the longer of the 2 lines from it was the bottom of the quadrilateral and then given the bottom 2 corners, estimate where the top corners were using simple geometry given that all the corners were for all intents and purposes right angles and given the sides of the quadrilateral were roughly 85% of the length of the top and bottom.

Even though the bottom 2 corners were very often correctly found by the algorithm, there was a problem with finding them in a few of the images, in that after tweaking the threshold for the corner finding algorithm it still sometimes returned a corner in between the 2 bottom corners, as in the left picture in figure 5.5. As the algorithm took the longest line from the bottom point to be the bottom line of the quadrilateral, it created a much smaller quadrilateral, filling only around a quarter of the space it should at the bottom right of the actual quadrilateral in the image. It was for this reason that we added steps 5 and 6 to the algorithm above.

Their purpose is to smooth the set of lines, so that if the bottom is split into 2 lines like in the example below, these can be merged into 1. This is done by first calculating the gradients of each line and then going through every set of 2 lines which have a common endpoint and comparing the gradients to decide if the lines should be merged. The measure used to compare them is the gradients are divided by each other and if the obtained value is between 0.25 and 4, the lines are merged.

Once the 4 corners have been estimated, they are put in the necessary ordering required for the homographic transfer and passed on to the homography code

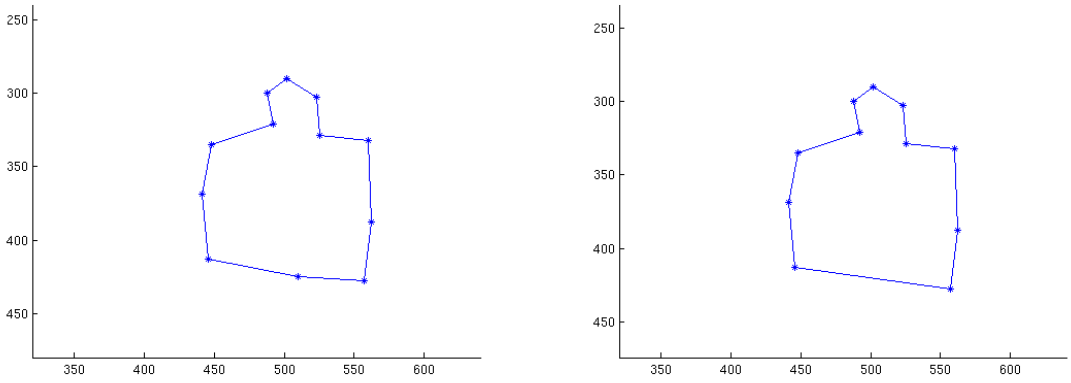


Figure 5.5: An example of the results of the corner finder before and after smoothing

which is the same as the code used for the earlier transfer except this code accepts the 4 target points as parameters and is called remapVideo.m.

6. Conclusion

7. Inclusion of Code

Bibliography

[1] Kinect website. http://www.xbox.com/en-US/kinect

[2] source 1: @articleFischler:1981:RSC:358669.358692, author = Fischler, Martin A. and Bolles, Robert C., title = Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography, journal = Commun. ACM, $issue_date = June 1981, volume = 24, number = 6, month = jun, year = 1981, issn = 0001-0782, pages = 381--395, numpages = 15, url = http://doi.acm.org/10.1145/358669.358692, doi = 10.1145/358669.358692, acmid = 358692, publisher = ACM, address = New York, NY, USA, keywords = automated cartography, camera calibration, image matching, location determination, model fitti$

[3] This website: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FISHER/RANSA Author: Bob Fischer Year: 2002 Accessed on: 22nd April 2012

[4] AV Lecture Note 1. www.inf.ed.ac.uk/teaching/courses/

av/LECTURE_NOTES/SMALL/avsys1s.pdf c

[5] Matlab's Image Processing Toolkit. http://www.mathworks.co.uk/products/image/