EPCC-SS-2001-09

MOCHA - Machine Vision in Java

Jay Bradley

Abstract

The aim of this project is to utilise cheap imaging hardware to produce a simple machine vision system using Java. This is achieved through the implementation of an application, MOCHA, which analyses coffee pots. Three areas of machine vision are addressed in the project: image capture and display, digital image processing and inference. Algorithms for manipulating digital images and inferring knowledge from them are addressed. The application is simple to use and its source code is simple to understand and to adapt for future development. This project does not address every area and approach of machine vision but provides a basis for more advanced work.

Contents

List of Figures

1 Introduction

This project [6] investigates machine vision in Java. All the algorithms discussed are employed in a simple application called MOCHA - Minimal Optical Coffee Height Analysis. The application is capable of detecting the presence of a coffee pot in an image and stating how full of coffee the pot is.

Machine vision hardware can often be expensive. The hardware needed is the biggest contributory factor. Usually the better the optical input is the better the analysis will be. This cost is not just for quality camera lenses but also for lighting systems. Also, machine vision requires the use of some sophisticated software for image manipulation. This again can be costly. This high cost prohibits many possible applications of machine vision. This project aims to lower the entry level for machine vision applications. In this project all code is written in Java; a high level, reasonably easy to understand language. Java also means portability. The application can be run on highend workstations or budget desktop computers. This is important as machine vision can be a very processor intensive task, particularly if real-time constraints are added. Java is readily portable and has excellent socket communications support so it should be possible to spread the processing over a number of workstations. This is a cheap but effective option to true dedicated parallel machines. This could reduce the setup costs and bring machine vision to a larger potential audience.

2 Background

Machine vision can be split into three areas: image capture, digital image manipulation and inference. This section gives a brief introduction to each area, as well as providing background detail on the Java programming language and resources used in the project.

2.1 Image Capture

Digital images need to be gathered for analysis. Many things affect the usefulness of the images. A poor input image will most likely result in a poor success rate for the identification of features in that image. Of course there are ways of improving image quality through post-processing but it is wise to give some consideration to the quality of the initial images. Although small details such as translation of an object or noise can be fixed or overcome by the machine vision software, the consistency of lighting conditions between images is of high importance. Different lighting conditions for the same scene can radically alter the information conveyed in the resultant digital images.

Certain conditions must be assumed for image scenes. For example if the lighting changes drastically (from day to night for example) it would be nonsensical to use the same processing chain [1] for both situations. This does not mean that different capture environments cannot be overcome. It is useful to record all the possible environmental factors that can change and devise strategies that can deal with them. As a simple example it is possible to remove some lighting differences by equalizing all of the images captured. This uses a histogram equalization function which is covered in Section 4.2.3. Usually this function spreads the colour intensities used in an

[1] A processing chain consists of one or more ordered image filters.

image evenly from black (zero intensity) to white (full intensity) allowing the different images to be compared meaningfully.

Some constraints are easily maintained. In most situations the camera's distance from the subject scene can be fixed. At the very least the distance is often known and can be compensated for with a zoom lens. The height of the camera, and therefore the perspective view of the image, can be easily fixed with a tripod or some other suitable mounting.

2.2 Digital Image Processing

Digital image processing techniques can be used for two purposes in machine vision. Graphical filters can improve the detail of an image. Edge detection operators can find and display the edges of objects in an image (Section **??**). Thresholding (Section **??**) and histogram equalisation (Section 4.2.3) can also be used to bring out important details in the image). Note that these graphical filters often provide more information by removing non-important detail from the images although in doing so they do alter the image. More important with regards to machine vision these filters and operators can also be used to make images more consistent with each other. For example a red circle and a blue circle may look very different at first but after thresholding and edge detection they can both be identified as belonging to the same group - circles.

2.3 Inference

After capturing the images and running them through any number of filters we still have no information about the contents of the image scene. Inference is the process of pulling information from an image. In this project there are two bits of information we are looking for: whether or not the image contains a coffee pot; and if so, how much coffee is in the pot?

2.4 Java

In an ideal world the code should be portable, easy to understand (though this depends greatly on the programmer), extensible and high performance. To best reach these goals, the coding is done in Java. Java is portable, reasonably easy to understand for any programmer and highly extensible through its application of object oriented paradigms. Is Java high performance? The more it is used in high performance applications such as this project the more it can be considered to be high performance! Improvements to the Java runtime environment and Java compilers are being made regularly. With widespread use in high performance areas comes improved compiler design which gives more efficient code. This application is not intended to be used in an industrial scenario so performance is a consideration; not a necessity.

2.5 Existing Resources

ImageJ [4] is a public domain toolkit for image manipulation written in Java. It allows the interactive application of image operators and filters. Many of the algorithms in this project have been ported from ImageJ. ImageJ uses its own classes to represent images and so some work was needed to make the algorithms work with standard Java image representations. Sometimes only a few lines needed to be changed but other times the algorithm needed to be written from scratch. Most often the image filter algorithms were amalgamated from descriptions in Digital Image processing [9] and HIPR2 [3]. Material on the Hough transform was found at CVonline [1] and in many brief mentions in a number of machine vision books.

3 Resulting Work

3.1 Amount of Coffee Analysis

A number of graphical filters are used to make the coffee appear as a contiguous area of black. These are, in order: auto level threshold, max thresholded image with the original, equalise twice, auto level threshold, median twice.

The area of black in the image is then calculated by simply counting pixels. The volume of coffee is calculated using a linear function from the number of black pixels in a full coffee pot to the number of black pixels in an empty coffee pot. In the current version of MOCHA, these numbers are hard coded into the program, though it would be much more useful to have them set by giving examples of full and empty coffee pots. It is assumed that the handle in the image will always stay more or less constant (at worst the handle could be out of sight if placed behind a full pot of coffee) so therefore, will not affect the readings.

It is worth noting a big mistake made in this area of the project. Initially some test pictures were taken of the coffee pot. For the purposes of filling the coffee pot without wasting coffee, the coffee level was raised by adding water. This had the affect of making the coffee less black. This meant that with the above sequence of filters, the coffee stood out as bright red afterwards. It was intended that the area of red would be counted and used in the linear equation. This means that it may not be neccessary to use so many image operators as are being used now. It is probably possible to get a similar image by using less proccessing steps. It is worth noting that coffee analysis would not work if somebody were to make weak coffee! Future generations of MOCHA could address this problem.

3.2 Coffee Pot Identification

The Hough transform can be used to detect features in an image. Its main strength is that it continues to work in the presence of noise in the input image. The transform requires a description of the features that it is attempting to detect. The classic Hough transform uses parametric descriptions of regular curves such as lines, circles, and ellipses. The classic Hough transform can be generalised to work with any arbitrary shape. In this project the classic Hough transform has been implemented with straight lines. The classic Hough transform is not of any use for the MOCHA application. The classic Hough transform was implemented only as a basis for expanding it into the general Hough transform.

As stated above, the classic Hough transform is of no use to the MOCHA application but to describe the general transform without first explaining the classic algorithm would be difficult. Also, the classic Hough transform could be useful in future applications of MOCHA and so has been left in the code.

Both Hough transforms work on binary thresholded images. These are images containing only black and white pixels. A white (actually non-black) pixel indicates the possible presence of a line (classic) or a shape (general) in the original image. Both Hough transforms work on a per pixel basis. For each pixel some processing is done. Thus it follows that the more white pixels there are in an image the more processing there is to be done. Because of this, it is usually a good idea to skeletonise the images before passing them into the Hough transform functions. This can reduce running time dramatically.

The graphical filters used in the processing chain should be used to enhance the image so that the shape or lines to be detected are shown as clearly as possible. If, with the best image processing, the resultant image is still poor then this may be overcome by not skeletonising the image before input to the transform function. This leaves more pixels in the image and gives a better chance of matching a shape in the general Hough transform and also, of finding real lines in the classic Hough transform.

3.2.1 Classic Hough Transform

The classic Hough transform can be used to detect the presence of straight lines in an image. Note that lines are infinite and that this transform does not detect line segments. Further processing would be needed to find the ends of the detected lines.

The code for the classic Hough transform is discussed in Section 4.4.1. This is a brief introduction to the ideas behind the transform.

Each pixel in the input image votes to say that it is part of all the lines that pass through that point. The voting is stored in an two dimensional array. Two ways of indexing the array were considered. Each element in the array needs to uniquely describe a possible line. There are two common ways of describing lines:

- $y = mx + c$
- $x\cos(\theta) + y\sin(\theta) = r$

Vertical lines cannot be represented with the first equation so the second, normalised equation, is used. If the first equation was used then the array would be indexed by m and c (the gradient and the y intercept). With the normalised equation lines are represented by θ and r.

Voting is done by adding one to the array value indexed by θ and r. It is possible to do greyscale voting where a vote adds 2 or more to the value and a lesser amount to the surrounding array values. Greyscale voting improves the accuracy of both of the Hough transforms but is implemented in the current version of MOCHA.

If we input Figure 1 into the classic Hough transform and then display the voting array we see Figure **??**. The Hough space wraps around at the left and right edges. Figure 2 shows four highlighted points. This indicates the highly likely presence of four straight lines in Figure 1. The Hough space array can be thresholded to remove non-likely lines and then transformed back into Cartesian space. This is shown in Figure 3.

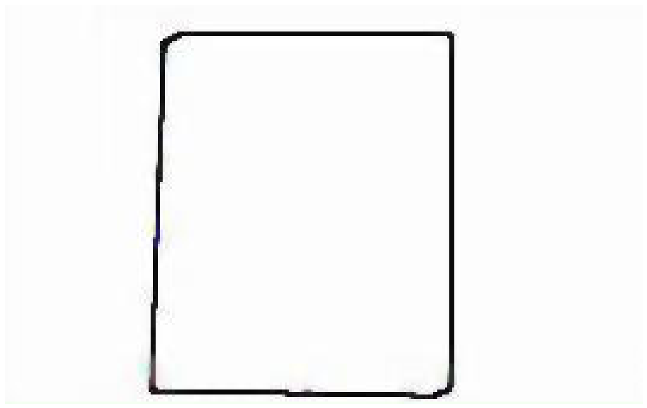


Figure 1: A box with four straight edges

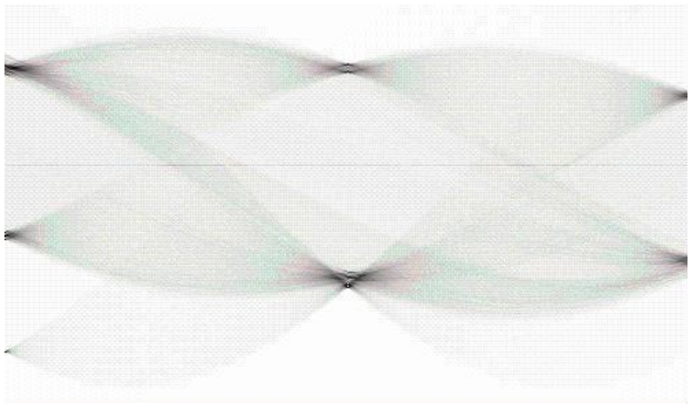


Figure 2: Classic Hough Space

3.2.2 General Hough Transform

The general Hough transform can be used to detect the presence of any arbitrary shape. Shapes are described below. At present MOCHA can detect translated and/or scaled shapes. It would require minimal extra work to implement the detection of rotated shapes.

The general Hough transform works in a similar manner to the classic Hough transform. Each pixel in the input image (much like Figure 13) votes to say that it may be part of a shape. The shape must be predetermined. See Section 3.2.4 for a description of shapes. Each pixel in the input image may be any pixel in the shape. Shapes are described using a reference point. So for each pixel in the input image and for every pixel in the shape it may be a vote is cast to say where the reference point would be. A vote is cast by adding one to the Hough space array. The general Hough space for non-scaled shape detection is shown in Figure 5. When using the general Hough transform without scaled shape detection, the Hough array is indexed by Cartesian coordinates. The array simply represents where (in two dimensional coordinates) reference points for shapes are likely to be. The Hough space can be thresholded as (for the classic Hough space) and the remaining points can be used with the shape description to map the shape(s) over the original image as in Figure 6. (Note that this is not implemented in MOCHA. The general Hough space points are mapped into an new empty image.)

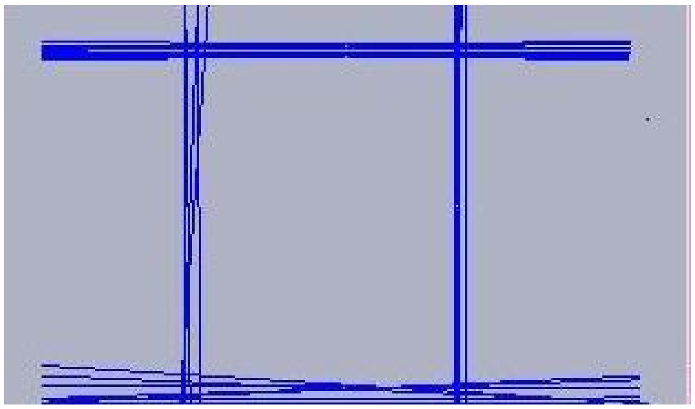


Figure 3: Classic Hough Space transformed back to image space

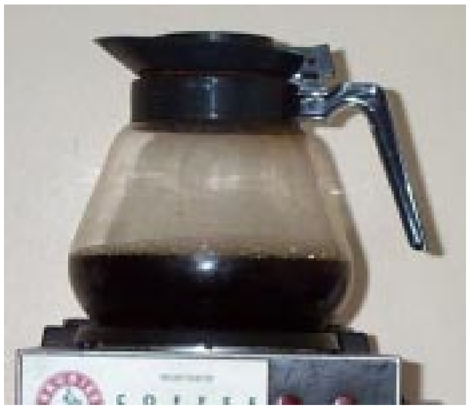


Figure 4: Coffee Pot Source Image

3.2.3 Detecting Scaled Shapes

It is possible to detect scaled and rotated shapes using the general Hough transform. To make the representation of shapes easier to scale and rotate the shapes can be stored as a series of polar coordinates. Polar coordinates are given by:

- θ : the angle between horizontal and the line from the origin and the point
- r : the distance from the origin to the point

To scale a shape stored like this all the r values must be increased or decreased together. Top rotate a shape stored in this way all the $theta$ values should be increased or decreased together. Note that the θ values must wrap around at 2π.

If scaled and rotated shapes are to be detected the general Hough space voting array should be made four dimensional. An extra dimension each for the scaling and the rotation. The size of these new array dimensions is dependent on the accuracy level required. As accuracy increases the memory required will increase. For example if rotation is to be detected 2π should be split up into a reasonable number. 360$\deg$!!!!!!!!!!!!!!! would give an accuracy the same as degrees. This means the rotation dimension of the general Hough space array would have to store 360 numbers. Similarly, to detect a shape twice the size of the stored shape the array dimension

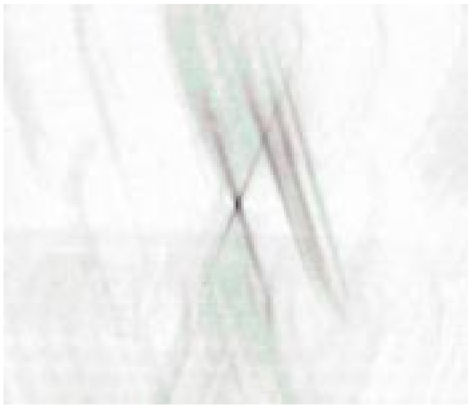


Figure 5: General Hough Space

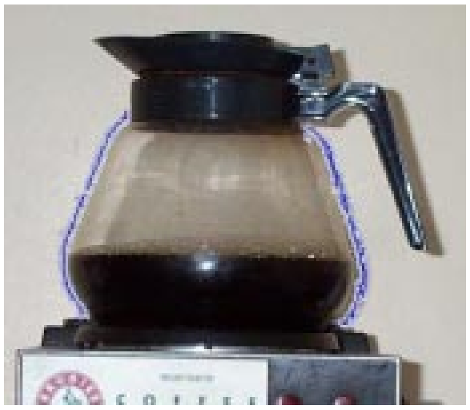


Figure 6: Detected Shape Displayed Over Source Image

would have to be capable of holding $2 \times Max(r)$. As the index of the scaling dimension increases so would the size of the r values of the possible shape.

3.2.4 Shapes

Shapes are stored as a list of points indicating the offset from a reference point. In this way shapes can be detected by translating the reference point. If the points are recorded in polar coordinates (θ, r) then scaling and rotating of shapes is made simpler.

4 User Manual

The application is written in Java and utilises AWT and Swing components [7] to build simple GUI. The main window is an extension of a Swing JFrame. Subsequent windows are instances of JFrame's containing a JScrollPane each. A JScrollPane is a very convenient container for displaying arbitrarily sized images. The image will always be displayed in a reasonable way using a JScrollPane. Each JScrollPane is assigned an ImageIcon. The ImageIcon's are constructed from a JLabel which contains the image to be shown.

All of the processing is done using AWT BufferedImage's. The pixel values can be got at by using the getRGB and setRGB methods of a BufferedImage object. Both of these functions work with one dimensional arrays of integers. The length of the arrays should be the picture's width multiplied by the picture's height. A Java integer is 32 bits long. This allows 8 bits for each colour (red, green, blue) and an 8 bit alpha channel. The alpha channel controls the opacity of a pixel and is not used in MOCHA. The alph channel is the most significant byte of the integer. Then follows the colours: red, green and blue in that order. With 8 bits you can represent 256 different levels of intensity for each colour channel. Full intensity (255) is often represented as 0xFF (hexadecimal) in the code.

4.1 Menus

Only the main window contains menus. There follows a brief description of the three main window menus.

4.1.1 File

The File menu contains the Load... command which does the main processing. It loads a JPEG file and performs a general Hough transform against it using the current shape template. This menu option is frozen out until there is a shape template loaded into memory. The File menu also contains the exit command.

4.1.2 Shape

The Shape menu contains all the commands for building and loading shape templates. The commands are: Build shape picture: This command asks for a JPEG image file and saves another JPEG image file. The saved image is made by applying several image operations and filters to produce a decent approximation of the shape contained in the source image. This image normally needs to be touched up in a separate paint program such as GIMP [2]. Load shape picture: A JPEG picture is loaded and a shape is made from the image and used as the current shape template. Load shape: This loads a file containing serialized objects of type PolarPoint. The array of PolarPoints is used as the shape. The shape is drawn and displayed in a separate window. Build shape: This loads a JPEG file and builds a shape from it. The shape is made from PolarPoints. The JPEG is binary thresholded before the shape is built. This is to remove any erroneous pixels created by JPEG encoding.

4.1.3 Help

Contains only the about menu item which raises a small dialog with brief information on the application.

4.2 Filters and How They Work

ImageJ [4] was used to try out potentially useful filters on sample pictures. Such a tool is useful in finding appropriate filters and operators interactively. Section 5.1 suggests the development of such a tool as an extension to the MOCHA package. As many filters and operators as needed to produce useful images should be strung together into a processing chain. A processing chain is just an ordered list of functions that the image will be passed through. Once a good set of filters was found in ImageJ the filter functions needed to be written for the MOCHA application. These functions are all contained in the ImageOps class as public static methods. Note that you do not need an instance of a class to call its public static methods. In this way a class is just a container for a number of related functions.

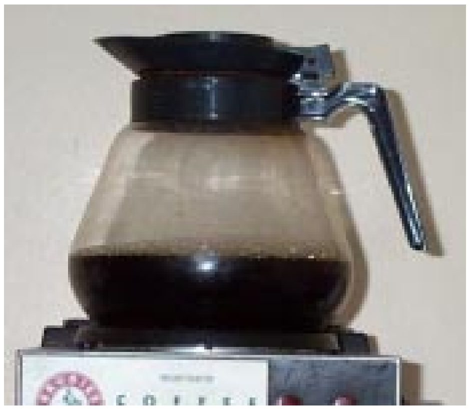


Figure 7: Source Image of Coffee Pot

4.2.1 Threshold - Figure 8

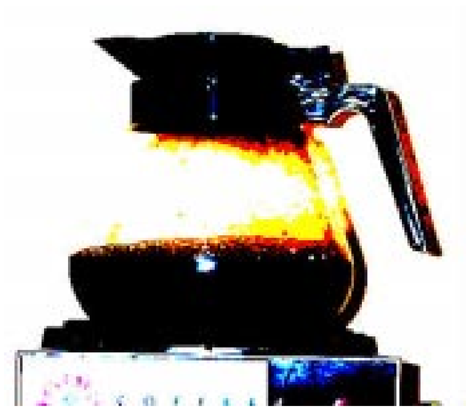


Figure 8: Threshold with automatic threshold level of Figure 7

Function input:	BufferedImage containing source image Threshold Value: A number between 0 and the maximum intensity level (255 for 32 bit ARGB representation)
Function returns:	BufferedImage containing thresholded image

Thresholding functions take a level that is used as a cut off point. Any pixel intensities below this point are removed from the image. There are a number of different functions for thresholding images in ImageOps. There are also some supporting functions. The only difference is that one (lookupThreshold) thresholds against each colour band separately instead of the average intensity. The getAutoThresholdValue function is very useful. It determines the optimum threshold value for the image it is given. The optimum value gives a balance between pixels removed and those left.

4.2.2 Max - Figure 9

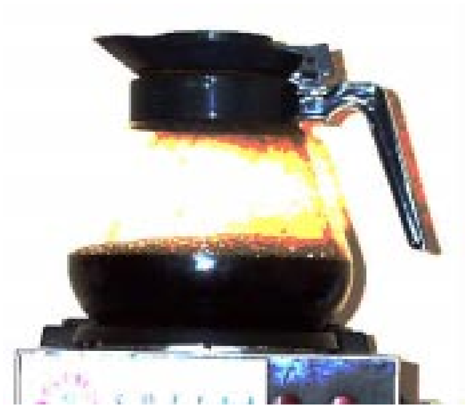


Figure 9: Max of Figure 7 and Figure 8

Function input:	Two BufferedImage's
Function returns:	BufferedImage containing max of the two input images or null

The max operator is the only operator implemented in MOCHA that takes two images as inputs. The function first test if the two images are the same height and width. If they are are not then the function returns null. It would be simple to build in functionality that works only to the smaller image's extents. In MOCHA the images are always going to be the same size so this will suffice. The algorithm moves through each pixel in turn. The maximum red, green and blue values from both image's pixels are found then combined into the new image's pixel.

4.2.3 Histogram Equalisation - Figure 10

Function input:	BufferedImage containing source image
Function returns:	BufferedImage containing equalised image

Equalisation functions work by spreading the colours of an image across the whole spectrum of colours available. The ImageOps equalisation function does not distribute the colours evenly. It is weighted by the square root function. This gives a different resulting picture that can sometimes be more useful. Histogram equalisation is often used to highlight features of the image that are too dark or too light to see with the human eye.

4.2.4 Skeletonise - Figure 11

Function input:	BufferedImage containing source image

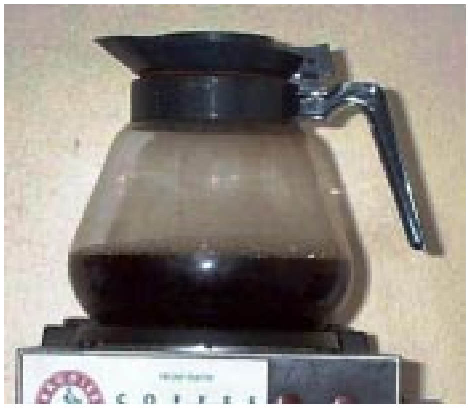


Figure 10: Equalisation of Figure 7

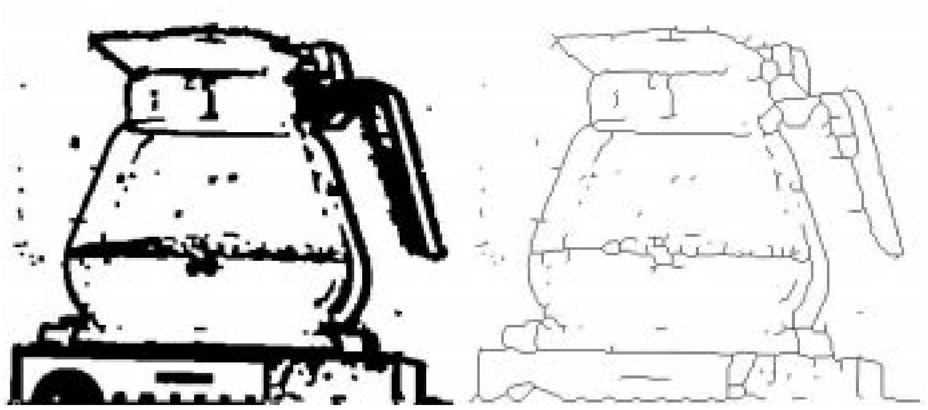


Figure 11: Skeletonisation in action

Function returns:	BufferedImage after skeletonisation

Skeletonisation is very useful as it removes potentially a lot of pixels whilst still maintaining most of the information in the image. It assumes the input is a binary image. Most of the work is done in the thin function which is repeatedly called by the skeletonise function. The algorithm stops when the thin function cannot remove anymore pixels from the image. The thin function works by stripping off pixels based on the possible permutations of the 3x3 neighborhood of pixels. Some permutations of the 3x3 neighborhood mean that certain pixels are more likely to be stripped off. There are more complicated but accurate algorithms described in Digital Image Processing [9].

4.2.5 Median - Figure 12

Function input:	BufferedImage containing source image
Function returns:	BufferedImage containing median filtered image

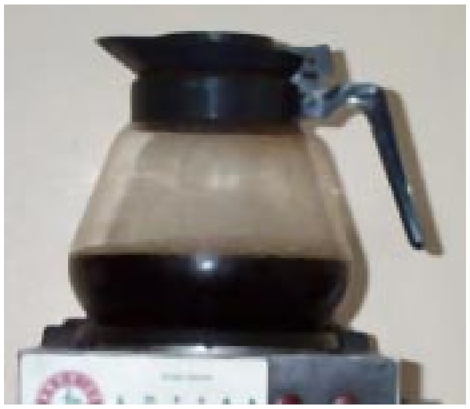


Figure 12: Median of Figure 7

The median function is similar but subtly different from a blur function. The median filter considers each pixel in turn. For each pixel it takes the median value of itself and the eight pixels surrounding it. If the filter had taken the mean of the nine pixels then it would produce a blur. Using the median makes areas in the picture "round off". If there are any isolated pixels then they are lost. Sharp points become more rounded. For this reason a median function is best used before looking at an area of one colour as it gives a truer representation of the area in the original scene.

4.2.6 Edge detection - Figure 13

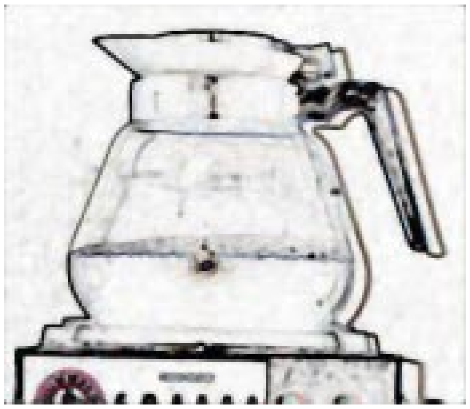


Figure 13: Edge detection of Figure 7

Function input:	BufferedImage containing the source image
Function return:	BufferedImage showing the edges indicated in the input image

There are two edge detection functions in the code. One uses a convolve operation with a kernel. The other is taken from ImageJ [4]. The functions give slightly different results. Use which ever gives the best results for the images you are using.

4.3 File Operations

All functions that handle the file input and output are contained in the FileOps class as static public methods. Functions for reading and writing JPEGS as well as reading and writing shape files are in the class. There are two functions for saving JPEG images. The standard methods for saving JPEG files as provided by Sun proved unreliant during this project. These are used in the saveBufferedImage function. The saveAsJpeg function uses a class from the ImageJ [4] function without alterations.

4.4 Hough Transform

All the functions for working with Hough transforms are contained in the HoughTransform class. The methods are all declared as public static.

4.4.1 Classic Hough

The classic Hough functions can be used to detect the presence of straight lines in an image. These can then be mapped back onto an empty image. See Section 3.2.1 for an explanation of how the classic Hough transform works. There are two functions needed to perform the classic Hough transform. First the input image should be passed to HoughTransform.classicHough which accepts the input BufferedImage and returns a BufferedImage containing the voting Hough space. The other two inputs determine how the voting array is scaled. Scaling is only important for displaying the voting array.

To map the points of the classic Hough space back onto an image the houghToLineImage function can be used. This function returns an image with the lines drawn into it (Figure 3). The function must also know the size of the original image that the Hough space was created from. The fourth input is a threshold value. A value of 0.5 means that the image will be thresholded with a value equal to half the highest vote. A value of about 0.8 is usually most useful. The lower the threshold the more erroneous lines are introduced.

4.4.2 General Hough

See Section 3.2.2 for a thorough explanation of the general Hough transform. The general Hough functions were derived from the classic Hough function implementations. Using them and the shape functions it is possible to detect the presence of any arbitrary shape in an image.

The shape functions can be used to build a shape using PolarPoints or (Cartesian) Points. These shapes must be passed to the both the generalHough function which creates and returns the general Hough space and the generalHoughToImage function which maps the Hough space back into an empty image which is returned.

The generalHough function also takes a threshold value. This threshold is a percentage and is used to determine whether a shape has been legitimately detected. The most voted for reference point in the Hough array must be greater than the percentage, equal to the threshold value, of the length of the shape. This ensures the test still works for large and small shapes.

5 Conclusion

The MOCHA application demonstrates the use of machine vision techniques without the use of any high cost high end equipment. A relatively cheap digital camera was used to capture input images. The code is inexpensive in terms of processing power and could be run on most home computers.

Java has been an easy language to learn [8] [7] and to work with for the use of image processing. With the advent of Java Advanced Imaging and Image I/O from Sun [5], Java should become more accepted as a language of choice for delivering high performance graphics tools. As a plus point GUI development is also quick and easy.

Opportunities for parralelisation exist in the MOCHA application. The general Hough transform does not execute fast enough to be run in real-time. The general Hough transform works on images which have a flat and simple memory layout. The Hough space voting array is also an uncomplicated storage structure. These two structures could easily be split up for work done by more than one processor. With JOMP [2], parallelisation of this code would be a worthy exercise.

The project has been a success in that it proved what it set out to do. That machine vision is not neccessarily a high powered highly complex task. It should be built upon to eventually deliver a full machine vision interactive toolkit.

5.1 Advancement of Project

- Add rotation to the general Hough transform. This is done by making the Hough space four dimensional and is discussed in Section 3.2.2.
- Look into alternatives to the Hough transform.
- Optimise the code. Most of the image filter algorithms and the Hough transform code could be further optimised, or possibly even parallelised as mentioned in Section 5.
- Implement an interactive image filtering program to arrange which and in what order the image filters should be used. The sequence of filters and operators could then be saved for use at a later date.
- Determine correct optimal characteristics for using the Hough transform. i.e. use skeletonise or not. It may be better to just thin images a little bit. Work to determine the optimal level for the matching threshold level will give more consistent results.
- Implement a calibrate function. This function would process a number of pictures that were definetely pictures of coffee pots. It would determine the smallest match threshold percentage to match all the images. It could even produce a shape for all the images. In this way, calibration for the Hough transform could be done quickly and simply.
- Take scale into consideration when calculating the coffee volume. For example a closer coffee pot does not contain more coffee. You would have to use the information from the general Hough transform do enable this.

[2] Java OpenMP. A standard code library for parallel programming.

- Remove coffee volume linear function limits (empty pot black pixels and full pot black pixels) from being hard coded. It should be possible to set them automatically by giving the program empty and full example images.

6 Bibliography

References

[1] Cvonline. http://www.dai.ed.ac.uk/CVonline/.

[2] Gimp. http://www.gimp.org/.

[3] Hipr2. http://www.dai.ed.ac.uk/HIPR2/.

[4] Imagej. http://www.rsb.info.nih.gov/ij/.

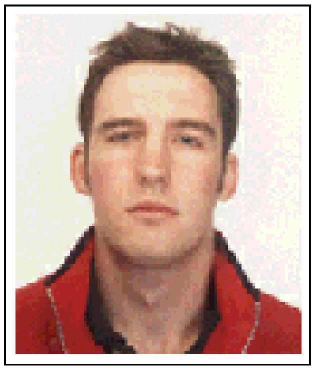
[5] Java. http://www.java.sun.com.

[6] Mocha project proposal. http://www.epcc.ed.ac.uk/ssp/2001/ProjectSummary/SS-2001-09.html.

[7] David Flanagan. *Java Foundation Classes in a Nutshell*. Third edition, 1999.

[8] David Flanagan. *Java in a Nutshell*. Third edition, 1999.

[9] Rafael C. Gonzalez. Digital image processing. Addison-Wesley, 1993.



Jay Bradley recently graduated with BSc (Hons) Software Engineering from the University of Newcastle. He is about to begin a PhD in Artificial Intelligence at Edinburgh University - Institute of Perception and Behaviour. September, 2001.

Supervisor - Neil P Chue Hong