Rubik's Cube Solver

William Pitt
c1015111
Professor Rosin
Dr Mumford
Bsc Computer Science
School of Computer Science and Informatics
03/05/2013

Abstract

The Rubik's cube solver consisted three main parts of the system, the identification of the cube, the calculation of the method for solving the cube and the physical manipulation of the cube to return it to a solved state. The image processing was based around the Hough Transform approach to identify the position of the cube as well as colour identification using brightness constancy. The Friedrich method was used to solve the Rubik's cube, the solution was always correct but not optimal. Lego Mindstorms was used to physically manipulate the cube, however the robotic part of the system did not work correctly

Acknowledgements

I would like to thank my supervisor Professor Rosin for useful discussion and guidance throughout the project. I would also like to thank Matthew Williams from the school's computer club for lending me a Lego Mindstorms NXT brick as the robot would not have worked without this.

Contents

Figures

1 Introduction

The overall goal of the project is to produce a device which can solve a Rubik's cube without the assistance of a human. This will be achieved by taking images of the cube to determine it's configuration, calculating the manipulations that are required to solve it and then manipulating the cube to a solved state by rotating robotic hands which are controlled by Lego Mindstorms.

Some terms are used in this report which are specific to this project and may cause confusion so they will be briefly explained. Whenever the term *manipulate* is used in this report it is referring to changing configuration of the cube by turning part of the cube whilst the rest is stationary, when the term *rotate* is used it refers to rotating the whole cube so that the configuration of the cube is unchanged but the cube's orientation has changed. A Rubik's cube is made up of 26 pieces called *cubies*. There are six *faces* of the cube and each face has a three by three grid of nine *squares*.

The system consists of three main sections, the first section is the computer vision part of the system, the aim of which is to calculate the position and configuration of the cube. The image processing section must be able to calculate the exact configuration of the cube from images taken by a web cam. The system does not need to successfully identify the configuration of the cube on its first attempt and can have multiple attempts to do so if required, however the number of attempts should be minimised to increase the speed of the system. The second section of the system is the cube solving portion, which must find a way of changing the configuration of the cube from its starting state to a solved state. The cube solver must be able to calculate a series of manipulations that will transform the Rubik's cube to a solved state. The robotics section of the system should be able to calculate and perform the physical manipulations of the cube that correspond to the steps which the solver section has specified. The robot should do this as quickly and as efficiently as possible, however this is not the author's particular expertise and there are limits to the improvements that can be made to the robotic system.

A hybrid of both the waterfall and spiral models was used during the development of the project. The overview of the aims and main goals were known from the beginning of the project and were unchanged during the process of the implementation. The project was researched predominantly at the beginning but further research was required partway through the project. The design of project was performed in small iterations and then implemented based on the designs. The project was redesigned and the implementation was changed if there were problems that occurred during the implementation of the project, but the key aims and goals remained unchanged. An example of the design adopting to fit problems that occurred during implementation is the addition of a second NXT brick when one was not sufficient, this is explained in greater detail later in the project.

2 Background

2.1 Lego Mindstorms

Lego Mindstorms is a technology which uses the constructive elements of Lego alongside the versatility of Java, with basic input and output devices around a processing unit called a NXT brick that together allow users to construct basic robotic systems. The NXT brick originally comes with Lego software installed on it, which restricts the user to a basic graphical environment which implements a drag and drop method of control. Lejos is an open source software suite which can be installed onto the NXT brick to allow users to write their own Java files, which can then be executed on the NXT brick. The NXT brick can be connected to a computer via USB or Bluetooth and messages can be sent between the brick and the computer whilst programs are being executed on both devices. A computer can also connect to several NXT bricks simultaneously if required. Motors can be connected to a NXT brick and can be used as output devices to manipulate the

environment. Signals from the motors to the NXT can be used to monitor the current status of the motor, such as the current speed and rotation. Standard Lego Technic parts can connect to the NXT brick and motors. This means that the Lego can be used to construct various structures and robots which can have basic input and output devices.[1]

2.2 Rubik's cube

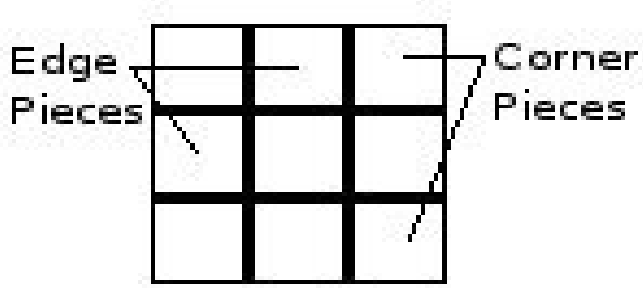


Figure 1: The Pieces of a Rubik's cube

The “Rubik's cube”, released in 1974, is a cube shaped puzzle which has been studied by mathematicians due to its complexity. The task of a Rubik's cube is to orientate all of pieces, or cubies, of the cube so that each side is a single colour. Each piece of the puzzle is unique and there is only one configuration that is deemed to be the solved state. (The overall orientation of the cube is irrelevant to the solution.) Each of the six faces of the cube can be rotated relative to the rest of the cube into three new positions (at 90^{O}, 180^{O} and 270^{O} to the original). At any time there are therefore eighteen different moves possible which will alter the configuration of the cube. The cube consists of eight corner pieces which have a visible square on three faces, with four corner pieces visible on each face. The cube has twelve edge pieces which have visible squares on two of the faces. There is a single centre piece on each of the faces of the cube which can not be moved from its original position, and will always remain in the same place relative to the other centre pieces of the cube. Rotation of these pieces does not affect the overall configuration of the cube. These centre pieces are linked around a central frame which is not visible without de-constructing the cube.

The concept of the puzzle is to 'shuffle' the cube by executing a series of random manipulations and then trying to return the cube to its original state. The cube must be shuffled by a series of random manipulations rather than dismantling the cube and then reassembling it, as very few reassembled configurations will result in a cube which is solvable. The cube must be solved by calculating a set of manipulations that will return the cube to its original state. There are many methods of calculating the manipulations that will solve the cube, where these methods will vary in terms of the number of manipulations and time taken to compute the solution. It is possible to calculate the solution with the fewest necessary moves, however a super computer and a large amount of time and money would be required to do this.[2] It is also possible to use very simple methods of solving the cube which require far more manipulations to complete the puzzle but require little processing power. The ideal solution is to minimise the combined calculation and manipulation times by finding a balance between the calculation and manipulation processes. The manipulation time is the time taken to physically move the cube and the calculation time is the time taken to identify a suitable series of manipulations.

The Rubik's cube that is being used for this project is quite new, as older versions of the cube had more friction between the moving parts and were difficult to manipulate. However the cube has been used many times for testing and so some of the visible surfaces have sustained damaged. This

makes it harder to identify the cube's components with the computer vision section of the system. So the vision system's algorithms must be more robust and effective at identifying false positives. An algorithm which can cope with a excessively worn cube should be able to identify many cubes with differing degrees of wear.

2.3 Sobel Edge Detector

There is a large amount of image processing involved with this project, the most basic of which is a Sobel Edge Detector, which uses a mask to identify changes of intensity in an image. The mask is a set of values in a grid, which is raster scanned across each of the pixels in the image. A convolution is a procedure which is performed on one function *a* by another function *b* to produce a new function *c* which is a modified version of the function *a.* A convolution of the mask at each of these grid positions is used to score the intensity of the edge at that point. This is performed with a horizontal and vertical mask which are used to calculate an "edge map", showing the intensity and the approximate angle of the edge at each point of the image. The equation that represents this resulting value *c* at the coordinates (x , y) on the image *a,* if a convolution is performed with the kernel *b* is:[4]

$$c(x,y) = \sum_{i=0}^{2}\sum_{j=0}^{2} b(i,j) * a(i+x-1, j+y-1)$$

The Sobel Edge Detector was selected as it is a simple to implement edge detection method which is efficient, robust and will provide an edge map which is adequate for the Hough Transform later in the project. Figure(2) is the mask used to calculate the vertical part of the edges.

1	2	1
0	0	0
-1	-2	-1

Figure 2: Sobel Mask

2.4 Hough Transform

The Hough Transform is a image processing technique for identifying graphical features of an image such as lines, circles and ellipses. The Hough Transform uses a separate accumulator space which represents values that are possible for the parameters of the features that are to be identified. A voting system is used to populate the accumulator where the greater values in the accumulator represent parameters of features which are likely to be present within the image.[3][4][8]

The type of Hough Transform which will be used in this project is for the identification of lines within an image. The identification of the lines starts by creating an edge map of the entire image. For each point in the image the value in the accumulator (the Hough Space), which corresponds to a line which can pass through that point is increased by the intensity of the edge at that point. The values in the accumulator are not simply incremented because the stronger edges found on the edge map should have greater influence in determining the presence of lines within the image. A local maximum will occur on the edge map where the parameters correspond to a line in the image. This

is due to several points of high intensity which lie on that line all voting for the parameters which correspond to it in the Hough Space. The Hough Transform was chosen as the method for detecting the Rubik's cube in the image as the the images were taken on a web cam and the Hough transform is not very susceptible to noise. A problem with identifying lines on the faces of the cube is that these lines can have breaks in them (at the edge of each square) which could impede other line detection methods. The Hough Transform is suited to this as it does not require these lines to be continuous for them to be identified. The gaps in the lines which are being referred to can be seen in the Figure(3).

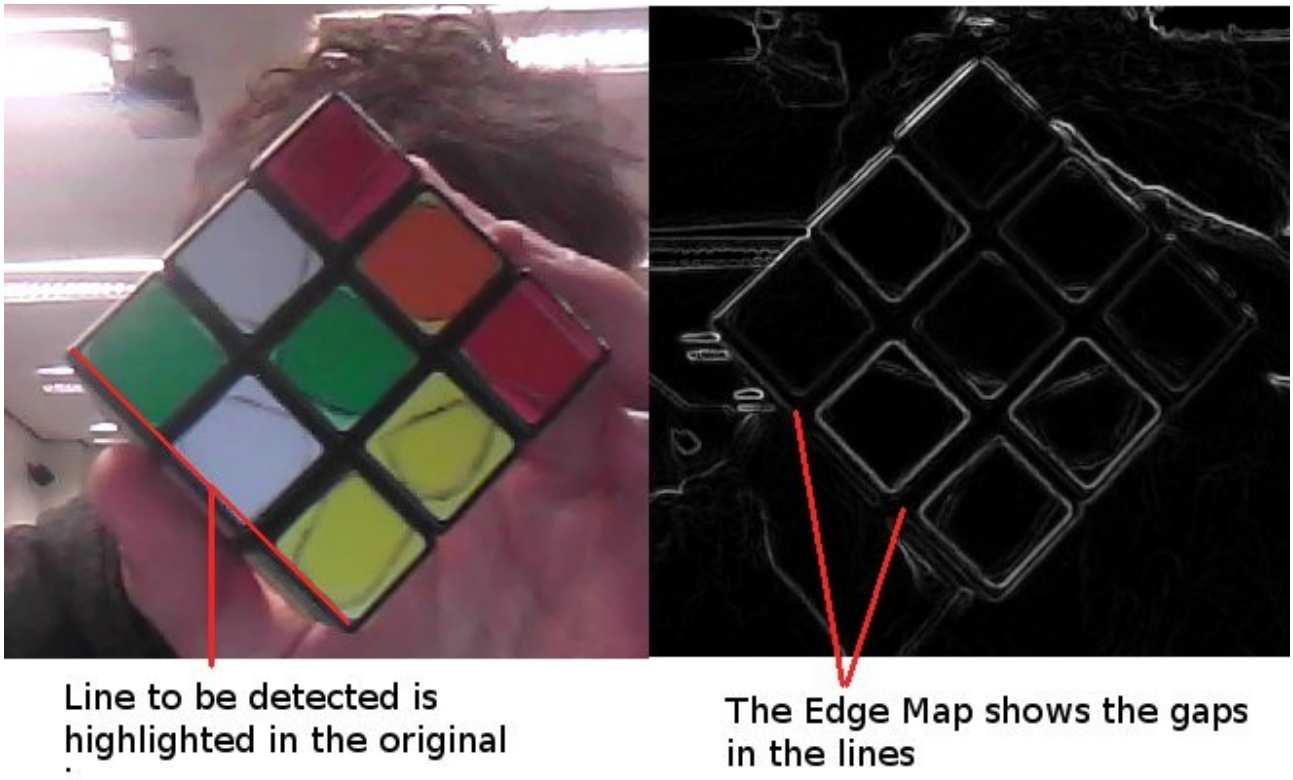


Figure 3: Gaps in the Lines of the cube

2.5 Template Matching

Template matching is an approach of identifying part of an image by using a template of a known object which could be in the image. The method used to identify the template in the image can be based on the strength of the features of the template. If a object has particularly strong features then the template can be based on the stronger features rather than the whole object. The template is raster scanned over the image and the difference between the template and the image at that location is used as a score. The lowest score represents the part of the image that most closely resembles the template. The template may need to be rotated or resized and can use a lot of processing power if not implemented efficiently. Template matching can be used alongside several techniques in computer vision which include edge maps and Hough Space. Template matching was chosen to be used in this project as it is a versatile method for identifying the specific features that were required for the project. The Hough Space had to be searched for a pattern that represented the Rubik's cube and template matching was suited to this as the template could be modified to search for different sized Rubik's cubes.[4][6][8]

2.6 Brightness Constancy

The light which is reflected off an object into the lens of a camera may not always reflect the true

colour of the object. The source of the illumination of the object can significantly alter the perceived brightness when viewed by a computer. Humans see colour differently to computers and their perception of colour is based on their surroundings. Brightness constancy is a technique which can be used in computer vision to estimate the lighting level that the image was captured in. Certain methods involve assumptions being made about the lighting of the object from features in the image and can then be used to calculate the colour of the object. Brightness Constancy was chosen for this project to improve the reliability of the identification of the configuration of the Rubik's cube as a constant level of lighting of the cube was difficult to maintain.[4][7]

3 Specification and Design

3.1 Basic Overview

The overall process of the cube solution is split into three main parts which all accomplish specific tasks, all of which are regulated by a central *control* class. The image processing classes which are involved in the system are the *camFeed, houghTransform* and *getColour* classes. There is a single class associated with the solving of the cube called *solver*, which takes a large amount of processing power and a large amount of the time spent on the project was used to create this class. The classes involved with the robotics of the system are the *adapterToRobot*, *onNXT* and *sendToNXT* classes. The *control* class is the class that is executed to run the system and commands the system. The *onNXT* class is uploaded to the NXT bricks and is executed on both of them. The *control* class is run and instantiates the classes that are required. This allows the *sendToNXT* class to create a connection with the NXT bricks at the beginning of the execution. There are two NXT bricks used in this system and a connection with both must be established from the beginning. The *camFeed* class creates a canvas that images can be displayed on, as well as an image grabber which is ready to take images using the web-cam. The *houghTransform* class is then used to track the position of the cube which is held by the user. The location and orientation of the cube is returned to the *control* class which is aware of the location to which the cube must moved for the hands to grasp it. Once the cube is correctly located the hands of the robot hold the cube using the *sendToNXT* class. The system then takes a series of images of each of the sides of the cube by using the g*rabImage* method in the *camFeed* class and rotating the cube with the *sendToNXT* and *onNXT* classes. Six images are taken of the cube, one image for each face of the cube. The images are stored in an array by the control class and the array is then sent to the getColour class as an argument.

The *getColour* class receives the images and must calculate the configuration of the cube. The *getColour* class uses a series of methods to calculate the colours on each of the sides of the cube and then uses the colours of the cube to calculate the configuration of the cube. The configuration data must be in a specific form which is recognised by the *solver* class. The *getColour* class must also check that the configuration of the cube is consistent (that it has the correct number of squares of each colour). If it is not consistent then new images of the cube are taken and the process is repeated. The *control* class then has to send the configuration of the cube to the *solver* class. The *solver* class uses a four step method of solving the Rubik's cube known as the “Friedrich Method”. The solver class then returns an array of strings which represent manipulations of the cube which, when executed, will solve the Rubik's cube. These manipulations refer to the type of rotation and the face of the cube that is to be rotated. The *control* sends the manipulations to the *adapterToRobot* class which must calculate a way of translating the manipulations of the faces into movements by the robotic hands. The movements of the hands which will be returned by this class must be described in terms of rotations and opening or closing of each hand. The *control* class must then send a series of moves from the *adapterToRobot* class to the *sendtoNXT* class. The moves are batched into sets of steps that the *sendToNXT* class can perform, between which the manipulation can be interrupted before the cube is fully solved. The *sendToNXT* class will communicate with the *onNXT* class to send the movements by the robotic hands to solve the cube. The *onNXT* class

receives the required movements and translates them into movements of the motors. After each batch of moves given to the *sendToNXT* class is performed, the *control* class checks that the cube is still correctly positioned by using the *houghTransform* class. If the *control* class is satisfied then it sends the next batch of moves to the *sendToNXT* class. A class diagram of this can be found in the appendices (Diagram 1).

3.2 Hough Transform

For the cube to be monitored during the execution of the system, the position, orientation and size of the Rubik's cube must be calculated within the image. The cube is difficult to identify due to the cube changing configuration when it is manipulated. This will lead to the colours of the faces of the cube changing, so it would be difficult to base the identification on colour. Template matching could be appropriate for the identification of the cube, however due to the large number of positions and orientations of the cube the identification would take a large amount of processing power and more efficient processes have been identified. The Hough Transform was a more appropriate method to tackle the problem as it can identify the lines of the image and then locate the cube within the image from the calculated lines.

3.3 Line representation

The standard way of representing lines in Cartesian coordinates is as y=mx+c where m is the gradient of the line and c is the y intercept. It is difficult to use this equation to represent lines for the Hough Transform as the value for m is unbounded. This makes it difficult to store the possible parameters of the lines in an accumulator.

An alternative method for representing straight lines in the Rho - Theta method. The Rho-Theta method uses a distance Rho and an angle Theta to represent straight lines. The Rho and Theta values of a line are based on a vector which is perpendicular to the line and passes through the origin. The angle that the vector is about the origin is the Theta value and the distance from the origin to the point where the vector meets the line is the Rho value. Figure(4) To calculate a line through the point (x,y) the following equation can be used Theta is represented by "θ" and Rho is represented by "ρ";

$$x \cos \theta + y \sin \theta = \rho$$

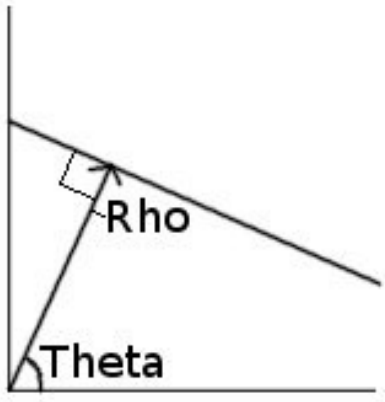


Figure 4: Rho Theta Representation

The values of Rho and Theta can be used as the parameters for the Hough Space and unlike the y=mx+c representation, a single accumulator is required to represent all of the possible parameters. The Theta value is limited to vary between 0^{o} and 179^{o} and the Rho value will be based on the size of the image, the Rho value can also be negative or positive as it can be either side of the origin.

3.4 Lines of the Rubik's cube

The array of squares on a face of the Rubik's cube produce two sets of parallel lines which are perpendicular to each other. The Rubik's cube is identifiable in the Hough space due to these parallel lines that are present on the face of the cube. The red lines draw in Figure(5) show the horizontal and vertical parallel lines which are present on the Rubik's cube. The lines can be split

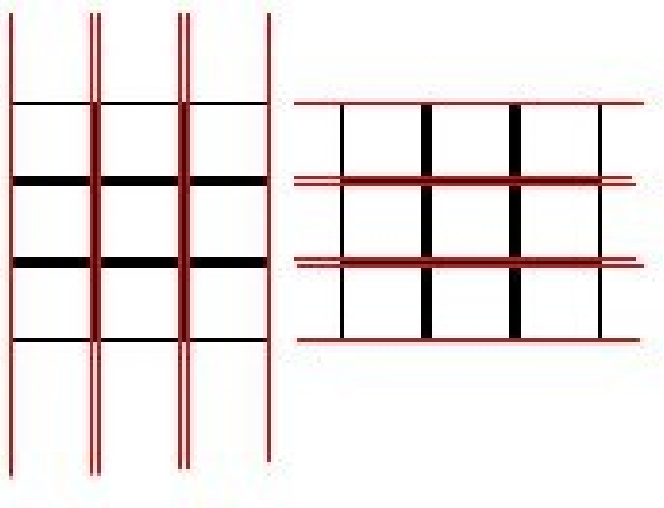


Figure 5: Parallel lines of the cube

into the two sets of six parallel lines when attempting to identify them within the Rubik's cube. The lines of the Rubik's cube are parallel and so can be identified in the Hough Space, as their Theta value, which is the angle from the origin, will be common for that set. The other set of lines is perpendicular to the first set and so the Theta value will differ by 90^{O}. Due to the Theta value being limited between 0^{O} and 179^{O} and the two sets of parallel lines differing by 90^{O} there must always be one set of parallel lines for which Theta is less than 90^{O} and another set for which it must be greater than or equal to 90^{O}.

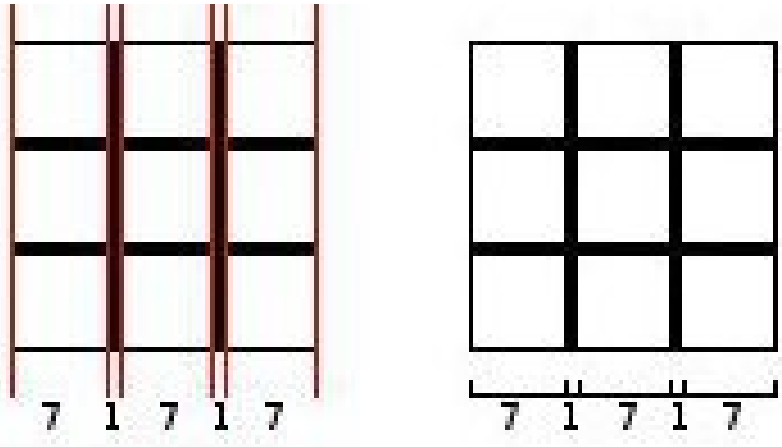


Figure 6: The Spacing of the lines of the cube

The distance between each of the parallel lines is not equal and will also vary due to the cube moving towards and away from the camera, so that the size of the cube image will change. The size of the gaps between the parallel lines will stay proportional to the other gaps when the cube is

moved. Figure(6) shows that the width of the gaps between the squares of the cube are seven times smaller than the squares of the cube and this can be used to identify the Rubik's cube within the Hough Space.

The system can sometimes struggle with the identification of the lines associated with the Rubik's cube in the Hough space due to noise. This is due to the poor quality of the images that are being processed. The hardware restricted the quality of the images as a web cam had to be used to obtain the images.

The number of unwanted votes that are accumulated in the Hough Space can be reduced to improve the clarity of the real lines within the image. The edge map that is calculated by the Sobel Edge Detector produces an estimation of the angle of the edge as well as the intensity of the edge. The system can use this estimation to limit the number of votes that are cast in the Hough Space. The estimation of the angle of the edge is not very accurate and so votes are cast for Theta value which differ from the estimate angle by less than 23^{O}. This will also have the benefit of reducing processing power required for this part of the program. A butterfly filter is one which is used to emphasise points in the Hough Space which correspond to correct lines in the image. The butterfly filter is raster scanned across the Hough Space and the value in the Hough Space is changed to a convolution of the filter at that position. The butterfly filter used in this system can be seen in Figure(7).

0	-2	0
1	2	1
0	-2	0

Figure 7: A Butterfly Filter

3.5 Colour Calculation and Cube Configuration

The colours of each of the squares of the cube must be identified so that the configuration can be calculated. The difficulty of this is that the lighting of the cube will depend on the environment in which the cube is in being viewed. The lighting will be difficult to control and so the system must be able to accommodate different levels of lighting.

The second part of the computer vision section of the system is the colour identification and finding the configuration of the cube. This is found within the *getColour* class which is given six images of the faces of the cube. The brightness of the pixels which represent the cube may change and so my program must acknowledge this. The method for identifying the colours of the cube began by calculating an average of the intensity of the six sides of the cube to estimate the strength of the illumination on the whole cube. This was used to estimate the strength of the light source to aid in the identification of the colours. The mean average colour for each of the squares of the cube was taken. Due to the condition of the cube the edges of the squares were not included when calculating the average colour. The average colour of each square was compared to a pre calculated set of colours to calculate the colour that it most closely resembled. The brightness calculated previously was used to assist in the identification of the colours by assuming that the illumination was constant across the object and normalising each of the calculated colours.

The colours may not be successfully identification in the *getColour* method and this could lead to the squares being assigned as the incorrect colour. The pieces of the cube are unique and so the system would become confused when trying to compute a piece which does not exist, this could lead to incorrect identification of the cube. A method is required to check the colours of the cube and attempt to correct them.

The number of squares of each colour are then counted to check that there are the correct number of each of the colours. If there is not nine of each colour the system attempts to resolve the problem by recalculating the assignment of some of the colours. The colours which appear too frequently are identified the squares that correspond to these values are re-evaluated and the colour which is most similar to a different colour is reassigned. This is repeated on each of the colours until there are the correct number of each colour.

The Rubik's cube *solver* class takes the configuration of the cube as a argument in a specific manner which is based on the orientation and position of the corners and edge pieces. The cube can also be represented as 54 squares with a value referencing a colour at each square. There must be a method which can translate the colours of the squares of the cubes to the unique position and orientation of each of the edge pieces.

The system then attempts to identify the configuration of the cube based on the colours that were identified. If the system can not resolve the identify the configuration then a new set of images are taken of the cube. The system then returns the configuration of the cube as arrays of the edge and corner pieces.

3.6 Solving the Rubik's cube

The Rubik's cube had to be solved by calculating a series of manipulations on the cube that change the configuration of the cube from its original state to it's solved state. The method used to produce the solution should minimise the number of manipulations of the cube and calculate the moves in a reasonable amount of time.

As the *solver* class part of the system was mentioned in great detail during the Interim Report, there is little detail in this section of the report, however a large amount of the time allocated for this project was spent in this area. The *solver* class is given the configuration of the cube in a fashion which was chosen to make it easy to understand and process. The Friedrich method is used by the *solver* to find a suitable series of manipulations for solving the cube. Once the solving algorithm of the cube is found all of the rotations of the cube must be removed, the *solver* class does this by changing the reference to the faces that are to be manipulated after this rotation . The solution is finally optimised so that any manipulations of the same face which are next to each other in the solution are concatenated into one move or removed if they are one complete rotation.

3.7 Connection to NXT and On NXT

There must be a connection active between the computer and each of the bricks for the full-length of the execution of the project, however messages will not need to be sent simultaneously to both of the brick as only one hand should be moving at once. The connection must also be terminated once the cube has been solved so that the NXT bricks can disconnect and properly end the execution of software that it is running.

The connection must be established between the computer and each of the NXT bricks, this can be done via Bluetooth. Bluetooth was chosen over USB for the connection from the computer to the

NXT bricks as it improved the mobility of the system and the USB cables may have interfered with the rotation hands of the robot. It has been decided that identical software is to be run on both of the NXT bricks, this is to simplify the system as it will make it easier to add more moving hands and NXT bricks to the system as well as requiring only one piece of software to write and update the devices. The program that is run on the NXT brick will consist of establishing a connection to the computer and then receiving, executing and acknowledging messages that will be passed to it. The execution of these messages will include rotating the motors attached to the NXT to move the robotic hands and closing the connection to the computer at the appropriate moment.

The hands of the robot will stay in the same position relative to the camera and this should not be changed each time the system is run. A default position is then created which is the variable which describe the location of the the cube when it is held by these robotic hands. The cube is tracked on the camera using the Hough Transform and the program will know the position of the cube relative to the camera. This default position can then be used when the cube is tracked in the cameras view to know when the cube is in the correct position. Once the cube is correctly positioned, the hands can then close on the cube so that it can be held ready to be manipulated. At certain intervals while the cube is being manipulated the movements of the hands must stop so that the system can use this edge detection and Hough Transform process to check if the cube is still correct.

3.8 Main Control

The classes had to be linked so that they could be used together, this was required for the system to function correctly but also allowed the parts of the system to be tested in conjunction before the final system was fully implemented.

The *control* class will be run by the user when the system is activated. The *control* class must make sure that all of the classes work correctly together, this will be particularly important when trying to take images of each of the sides of the cube at the beginning of the program. The *control* must tell the *getImage* class to take a image and then instruct the *sendToNXT* class to make a specific rotation of the the cube several times. The *getColour* class is then given the images of the cube and calculates the configuration of the Rubik's cube. The *control* class then gives the configuration of the cube to the *solver* class which calculates the movements required to solve that cube. These movements are passed to the *adapterToRobot* class which must work out the optimal method for rotating the hands in a manner which will solve the cube.

3.9 Physical Lego design

A simple design for this is to have two robotic hands perpendicular to each other, where both are able to grasp and rotate a face. The robot will be able to move the cube around so that it can access each face by having one hand open while the other is closed and rotating the closed hand. The robot will be able to manipulate the cube by moving the face require to be manipulate to a position where it can be grasped by a hand and then closing both of the hand and rotating the hand with the face that is required to be manipulated.

The most significant problems that had to be overcome when using this design is that when a hand grasps the cube it must not block the actions of the other hand. The way that a hand holds the cube must be acknowledged when assessing this problem. For the cube to be manipulated with two hand perpendicular to each other the cube can not hold any of the squares of the manipulated face as it would block the other hands rotation. Figure(8) Therefore the hand must only grasp the central square of the face that is being held. Each hand can be rotated to be aligned with two different axes and they both share a common axis which for this project is named the vertical axis. The hands can still block each other if the are both in a vertical position and both closed and so my program must

acknowledge this and ensure that it never attempts to do so. Figure(9) If one hand is vertical and the other is not then the hand that is vertical will block the other from manipulating the cube, however the vertical hand in this situation will still be able to manipulate the cube. Figure(10)

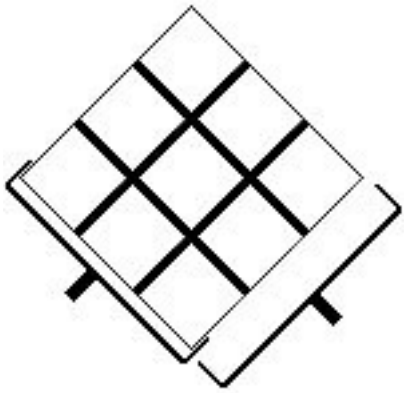


Figure 8: The cube being held. Hand A on the left Hand B on the right

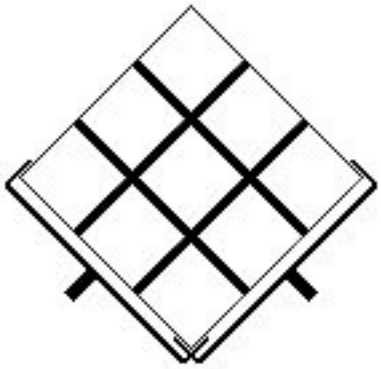


Figure 9: The hands block each other and so can not be positioned like this

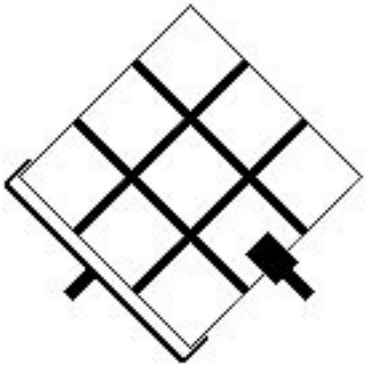


Figure 10: Hand A can be rotated to manipulate the cube. Hand B can not rotate

The design of the robotic hand has a cable which is attached to a motor which opens and closes the hand and this motor is rotated every time the hand rotates to manipulate the cube. This results in the cable being twisted whenever the hand is rotated which limits the maximum number of rotations of a hand in one direction. To overcome this problem the program must remember the rotation of the hand in specific direct and whenever possible rotated the hand towards its original position without slowing the performance of the robotic hands. The program does this by understanding when the direction of the rotation is irrelevant such as when the hand is being rotated 180^{O} and rotated the hand in a manner which will decrease the angle of rotation from its original position.

3.10 Adapter to robot.

Once the cube has been solved by the *solver* class a series of moves which refer to manipulations of the six faces is produced. There are three different positions to rotate a face from its original position using 90^{O} rotations. The two hands of the robot can be rotated, opened and closed. Each face of the cube is not held directly by a hand and so the robotic hands must reposition the cube before a face can be manipulated. The hands are also limited in their movement as they rotate in a specific direction due to the restrictions caused by the cables. The adapter must also ensure that the hands do not block or crash into each other.

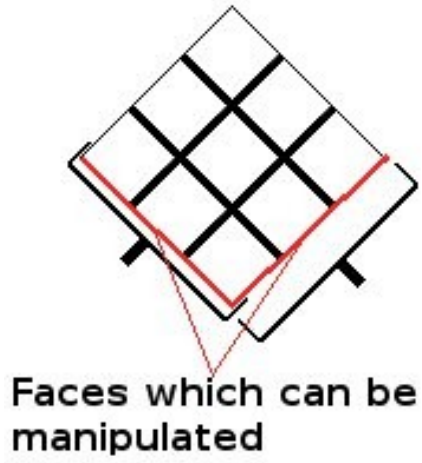


Figure 11: Faces which can be manipulated

The purpose of the adapter is to calculate the moves that are required by each of the robotic hands to perform the manipulations. The adapter must first move the required face to a position where it can be manipulated by one of the hands. For a face to be manipulated it must be at the bottom of the cube as show in Figure(11). The adapter must always know the position of the cube faces relative to the hands. The adapter can then identify the position of the required face and rotate the face to the correct position so the manipulation can then be performed. This is repeated for each manipulation and the next iteration begins by identifying the following face to be manipulated. The output of the class will be a set of moves which are based on rotations and grasps of the two hands.

The final large problem that was originally not acknowledge but had to be adjusted to, was the way that the NXT brick is designed. A NXT brick has seven ports the project required four ports, two for each hand, however only three of the seven ports are output ports which the project requires to power the motors. This resulted in the project needing two NXT bricks rather than just one with each NXT brick controlling a hand which are connected to the computer using Bluetooth.

The complexity and quality of the Lego hands is limited to the Mechanical Engineering ability of the author as well as the physical limitations of the Lego motors and the limited number of Lego parts that the author owns. This resulted is a limited but functioning design of the hands which could be improved upon by using more durable materials which were specifically manufactured for the project as well as using a greater number of robotic hands so that the cube can be manipulated from more sides, this would be maximised at five, as six would impede the access of the cube for viewing by the camera.

The accuracy of the motors in the Lego Mindstorms kit was unclear and was required to be tested as a small degree of inaccuracy could cause drastic problems later in the process. Inaccuracy in the motors could cause the cube to be accidentally dropped or the rotation of the cube to be wrong and jam the machine. The method that was used for testing the accuracy of the motors was to perform multiple 90^{o} rotations of the cube using the robotic arms, as this is the way that the robot would be used in the final system. The motor was accurate enough as after two hundred rotations the robotic arm remained in the original position and so if the other robotic hand was present hypothetically it could manipulate a face of the cube. This shows that the accuracy of the motors of the Lego Mindstorms is sufficient as they will be used less that two hundred times in a standard solution. The process was repeated five times to ensure accuracy of the testing. The time was recorded for the rotations of the cube and the average time taken to perform one manipulation of the cube was 0.87 seconds

4.0 Implementation

4.1 Camera Feed

The “camFeed.java” class uses google open source library to take images using the web cam of the computer that the software is being run on. The class uses a *Framegrabber* class to grab the image by simply creating an instance of it and using a *grab* method. The result is a *BufferedImage* which is a standard java class, this is returned whenever the grabImage method is called within *camFeed*. The *camFeed* class also has a canvas which can be used to display the image, this can be used to display live images of the cameras view which will can be updated to show the cubes position.

4.2 Colour Calculation and Cube Configuration

The Java class that is used for calculating the colours of the image is called *getColour*. The class is given images of the six faces of the cube as *BufferedImage* objects. The RGB value of a pixel is the Red, Green and Blue components of the light that has entered the camera, a larger value indicates that light of a higher intensity has entered the camera at that point. The class is given the position and orientation of the cube and so the exact location is known. The method *getColoursFromImage* is executed and calculates the average RGB value of each of the squares of the cube. The overall illumination of the cube is calculated by cumulating these values. The whole cube is used to calculate the average illumination, this is due to a single face may be lacking in colours of high or low intensity and so the results of the illumination of that face of the cube would be skewed.

The *calcSquare* method is then run to calculate the colours of each of the squares of one of the faces of the cube. The RGB values calculated for each of the squares in the previous method is

normalised by factoring in the average brightness of the cube. The RGB values of the squares are then compared with a pre-set value for each colour. The colour which most closely resembles the square is then assigned to that square. The method returns an array of the colours that it believes each of the squares corresponds to. The pre-set values for each of the colours was calculated by taking images of the cube under varying degrees of light. The colours of the squares on the face of the cube were know by the system and so hundreds of images could be taken which could then be normalised and the average values could be calculated.

The *fixColours* method was then used to identify when there were too many squares of one colour. The method identifies any colour which has too many squares assigned to it and attempts to rectify the error by calculating the square which most closely resembles a different colour. The method is run until there are nine of each of the colours or it has attempted to correct the squares unsuccessfully one hundred times. The method can not guarantee that the squares can be correctly identified and new images of the cube are taken if it fails to find the problem. The method may not identify a problem if two of the squares have been badly identified but there are still nine of every colour. This problem will be amended later by the *cornerAndEdges* method and new images will be taken.

The *cornerAndEdges* method is used to identify the configuration of the cube by calculating the unique corner and edge pieces based on the colours of the squares that it is given. The method contains arrays of the unique pattern of each of the eight corners and the twelve edges. For each of the corners of the cube the method calculates the three colours that are present at that corner according to the array of the colours of the squares. The three colours are then compared with each of the eight corner pieces to identify the orientation and position of each of the corners. This is repeated for the twelve edge pieces to find the overall configuration of the cube. If a corner or edge can not be identified then an error must have occurred previously and so new images of the cube must be taken.

4.3 Hough Transform

The *HoughTranform* Class begins with a simple Sobel Edge Detector which produces an edge map of the intensity and angle of the edges of the image. The intensity *i* and angle α can be calculated from the gradient of the horizontal Gh and vertical Gv components of the edges using these equations:

$$i = \sqrt{((Gh \wedge 2) + (Gv \wedge 2))}$$

$$\alpha = \arctan(Gv / Gh)$$

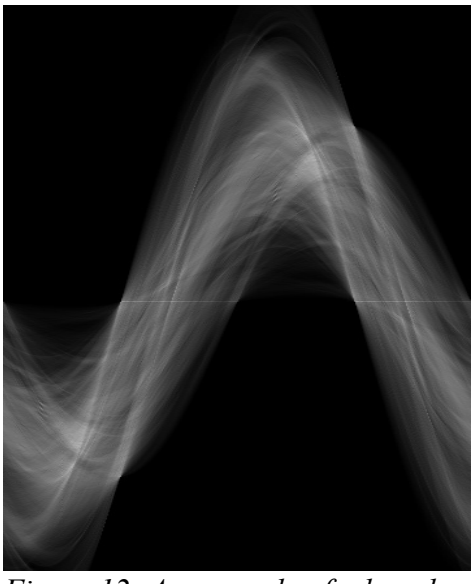


Figure 12: An example of a hough space

The Hough Space is then created, an example of which can be see in Figure(12). To improve the speed of the system without loosing too much accuracy to be able to identify the cube, the Theta values, which were on the x axis, were limit to 180 different values so that one each x coordinate represented 1^{O}. The possible Rho values were limit to the size of the image and each integer value of Rho was represented by a value on the y axis. This was calculated by using finding the distance corner of the image to the opposite corner using Pythagoras' Theorem. The Hough Space was accumulated by raster scanning across the edge map, if an edge intensity is greater than 500 then the point was used. The threshold of 500 was selected as the true values of the lines of the Rubik's cube had edge intensities greater than that value. The use of a threshold reduces the number of unwanted values that are accumulated in the Hough Space, this will reduce the required processing power and increase the visibility of the correct lines. The positions corresponding to lines through the point that varied from the estimated angle by less than 22^{O} were increased by the intensity of the edge map at that point. For a point on the edge map with coordinates (x, y) the values in the Hough Space with the coordinates (θ , r) are increased by using the equation:

$$x \cos \theta + y \sin \theta = r`$$

Once the values of the Hough Space has been accumulated the butterfly filter is raster scanned across the Hough Space and the value in the Hough Space is changed to a convolution of the filter at that position.

4.4 Template Matching within the Hough Space

The program must identify the position and orientation of the Rubik's cube by attempting to find the lines that refer to it within the Hough Space. With little knowledge of Hough Transformations the Rubik's cube does not appear to stand out in the Hough space. The Hough space can be noisy so the method for identifying the Rubik's cube must be robust. Single lines are fairly easy to identify within the image as the higher values in the Hough space represent the lines in an image. The position of the value in the Hough Space will correspond to the two variables Rho and Theta. However, it is far more complicated to identify a Rubik's cube within a Hough Space as the twelve lines that represent a Rubik's cube must first be identified. A single line in the image could represent a line of the Rubik's cube, but without the context of the other lines a system could not tell if it is part of the Rubik's cube. There are four variables that represent the position and orientation of the Rubik's cube. The *xOffset*, *yOffset*, the *angle* about the origin Theta and the *size* of the cube

The twelve lines of the Rubik's cube can be represented by their Rho and Theta values. As previously mentioned, the lines that can be used to identify the Rubik's cube are structured as two sets of six parallel lines, where the sets of lines are perpendicular. This means that for each set of parallel lines the Theta values are the same, the sets of lines differ by 90^{O} from each other. The Theta value of the first set of lines identified is represented by the *angle* variable and can be increased by 90^{O} to find the Theta value of the other set of lines. This means that one of the two sets of lines must have a Theta value less than 90^{O} and this set of lines will be identified first.

The *size* variable represents the scale of the cube by changing the distances between the lines of the Rubik's cube; the distance between each line is not equal but all will be scaled equally given the *size*. The *size* value that specifies the distance between the lines will be the same for both of the sets of lines. An array of the ratios of the distances between the Rubik's cube lines was used to calculate line distances, by multiplying the value in the array by the size. This *ratioArray* was essential in the calculation of the cube position and appears in the system as approximately (0,7,8,15,16,23).

The *xOffset* marks the lowest Rho value of any line within the first set of parallel lines. The *yOffset*

marks the lowest Rho value for the second set of parallel lines. The Rho values can then be calculated for the other lines of the cube by using the offset, size and *ratioArray*. The Rho and Theta values of the *xth* line of the Rubik's cube will be equal to:

For the first set of lines.
Theta = *angle*;
Rho = *xOffset* + *size***ratioArray*[*x*];

For the second set of lines.
Theta = *angle*+90;
Rho = *yOffset* + *size***ratioArray*[*x*];

The system must calculate the *size, angle* and *xOffset* of the first set of lines in the Hough Space, this is achieved using a Template matching based approach to identifying the values. Templates are created for the six Rho and Theta values that correspond to varying *size*, *angle* and *xOffset* values.

4.5 Identifying the First set of parallel lines

The Hough Space is repeatedly raster scanned by a template which increases in size with each iteration until the maximum *size* value is reached. The maximum *size* value was approximated by viewing the cube with the web cam and testing the ranges of the *size* of the cube based on the camera's field of vision. The *angle* value begins at 0^{O} and iterates through to 89^{O} in 1^{O} steps, as the first set of lines must have a Theta value between 0 and 90. The *xOffset* is incremented from 0 to the height of the Hough Space minus the height of the cube, as the whole cube must be included in the image for it to be identifiable. The height of the cube is calculated by multiplying the current *size* by the final value in the *ratioArray* (i.e. 23), which refers to the line of the cube with the largest Rho value. A score is calculated from the sum of the six points of the template. The *size*, *angle* and *xOffset* values that produce the largest score (and therefore the best match) are recorded.

4.6 Identifying the Second set of parallel lines

The *yOffset* must then be found within the Hough Space by creating a new template in which the *size* and *angle* values will not vary as this template is moved across the Hough Space. The Theta value is set to 90 + *angle* from the previous best recorded score. The *yOffset* is then incremented from 0 to the height of the Hough Space minus the height of the cube. The y*Offset* value that produces the largest score is recorded. This will determine the position of the Rubik's cube if there is little noise in the image.

This approach was tested several times and it was uncommon for the position to be identified correctly. To improve on this the template of the Hough Transform was therefore blurred to include the nearby values at each of the points originally calculated, but reducing the influence that they have on the final result. This is to accommodate the template not matching each of the lines of the Rubik's cube exactly.

4.7 Improving the Template Matching

The greatest score was often produced when one of the lines of the template correlated with a single strong edge in the background. So a disproportionately strong line could swamp values for the other lines. This problem was tackled by restricting the total scores which were recorded, discounting those where all six of the sub scores did not contribute significantly (providing at least 8.4% of the total).

Occasionally the system would still incorrectly identify the cube due to noise in the Hough Space

and so, to test the system, the ten greatest scores produced by the identification of the first set of parallel lines was recorded. The template that produced the greatest score would often match the lines of the cube. When the ten highest scoring templates were analysed it was almost certain that one of them would correctly identify the lines of the cube.

The method for identifying the second set of parallel lines had to be modified to accommodate the changes to the previous method. The method had to be repeated with the variables of each of the top ten scores from the previous method. The twenty five overall greatest scores were recorded so that they could be analysed.

4.8 Back Project onto Edge Map

The scores that were calculated by the system were then back projected onto the edge map to analyse the results produced when identifying the parallel lines. To do this a template was created that reflected the edges of the cube on the edge map. The template was position and orientated in accordance with the results. The system then counted the occurrences of strong edges that mapped to a point of the template. A value could then be return that represented the percentage of the points of the template that mapped to strong edges. Figure(13)

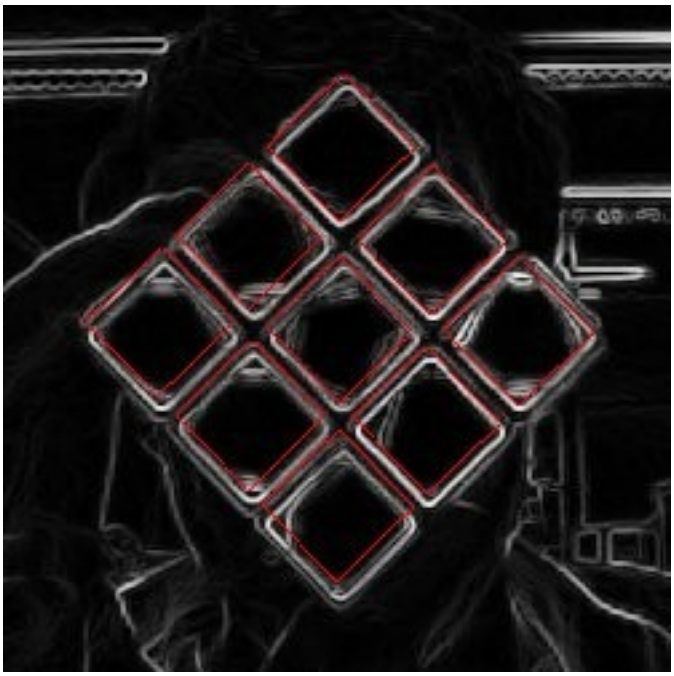


Figure 13: The template is back projected onto the Edge Map

4.9 Adapter to robot.

The adapter to robot class must convert the moves that have been calculated in the solver class into manipulations of the robotic hands. For each hand variables are used to represent the orientation of each hand, whether it is open or closed and overall rotation from its original position. The moves from the solver class are in the format of an array of strings with each variable in the array representing a manipulation of a face of the cube. The letter in the string represents the face of the cube that is to be rotated. The character after this letter is the type of rotation that is to be done on the face of the cube. A letter on it's own represents a 90^{O} clockwise rotation, a “2” following the letter represents a 180^{O} rotation and a “ ' ” following the letter represents a 90^{O} anti-clockwise rotation of the face . For example a 90^{O} clockwise rotation of the left face would be “ L ” and a 90^{O} anti-clockwise rotation of the down face would be “ D' ”.

The class must keep track of the positions of the faces of the cube and will do this by keeping an array of the positions and updating it whenever the cube is rotated. Whenever a hand is rotated one of three different things could be occurring, a manipulation of a face, a rotation of the whole cube or the hand is being readjusted. If both hands are closed whilst a rotation occurs the cube is being manipulated, if the hand being rotated is closed whilst the other is open the cube is being rotated, if the hand being rotated is open whilst the other is closed then the hand is repositioning so that it can hold the cube. This is calculated every time a rotation of a hand occurs so that the position of the cube can be updated. After a rotation the class must update the orientation of the hand so that it knows when a hand is vertical or not and the overall rotation must also be updated. The class will need to ensure that the hands are nether both open at the same time, must never be closed whilst both are in the vertical position and must reduce the overall rotation in a given direction whenever possible. The direction of the rotation of the hand will not matter when the hand is being rotated 180^{O} and also when the hand is open and is being repositioned between vertical or non-vertical. This can then be used to try to reduce the overall rotation from the original position.

The cube is orientated with two faces at the bottom, both at 45^{O} to the ground, these are the faces held by the hands. Once the required face and move are calculated the adapter to robot class must rotate the required face to a position where it can be manipulated. It does this by having the two hands positioned at the bottom of the cube and rotating the required face of the cube to the bottom. For a face to be manipulated the hand which is not holding the manipulated face must not be vertical as it would get in the way of the manipulation. This must be acknowledged when moving the face to be manipulated to the bottom of the cube. The face that is required to be manipulated can be in one of six different positions, or three pairs of positions, the top, the sides and the bottom. Figure(14) The face will not need to be moved if it is already in the bottom position. The remaining four positions are two pairs of positions which mirror each other. What is meant by this is the movements required by the hands to get the top face to the bottom will be the same for both of these faces but the movements will be done by the opposite hands. For instance to get the top left face to the bottom right position a 180^{O} rotation is required by the left hand. This is mirrored for moving the top right face to the bottom left position as a rotation of the right hand is required. This can also be applied to the repositioning of the faces on the sides of the cube as they too are mirrors of each other.

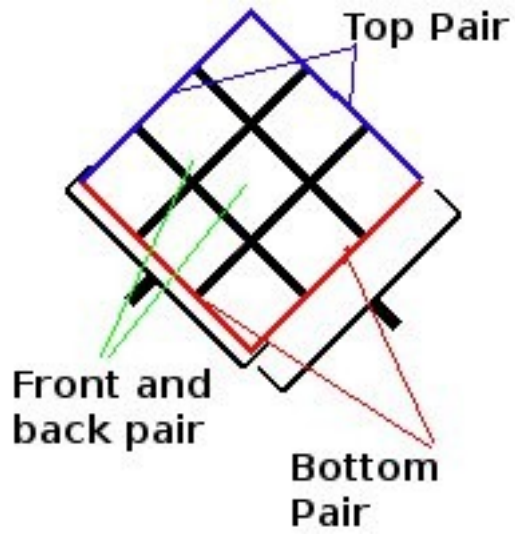


Figure 14: The Pairs of faces for the Adapter

4.10 Send to NXT and On NXT

It was difficult to get the correct syntax for this part of the system and so small snippets of code were taken from the website[6].
The *onNxt* class of the system waits for a connection to be established with the computer and uses *BufferedReader* and *BufferedWriter* methods to communicate with the computer. The class waits for a message to be sent, the message is interpreted as a move which is executed or a command to stop the connection.

The s*endtoNXT* class creates a Bluetooth connection with a NXT brick initialises *BufferedReader* and *BufferedWriter* methods in the constructor. The moves can then be sent to the NXT bricks using a method in the s*endtoNXT* class.

5.0 Results

5.1 The Hough Transform

The Hough Transform class involves identifying the cube within the image and the ability of the identification needed to be tested. The variables which represent the orientation and position of the cube are calculated in the Hough Transform. These variables are back projected onto the edge map and a score is returned which represents the similarity between the back projection and the edge map. To calculate whether the Hough Transform has correctly identified the cube within the image the score compared with a pre-set threshold of 70 which the score must be greater than for the result to be considered a success. To test the Hough Transform the cube is held in front of the camera and the program tries to identify the cube 1000 times, the time taken to run this was recorded. This was run three times with varying degrees of light. It should be noted that the back light from the screen casts some light on the cube. The cube was being held in front of the screen by hand and so the cube would move around slightly, this is how it is to be used in the system and so is a useful way of testing the class.

	Low Light	Standard Light	Bright Light
Success Rate	78.30%	82.40%	75.40%
Time Taken (s)	768	762	766

The result of this test showed the cube was on average correctly identified 78.7% of the time and took 765 seconds to run the tests. This test was repeated with no cube visible in the camera this resulted in none of the one thousand iterations identifying a cube when it was not present. The time taken for one attempt at finding the cube t and the probability of finding the cube p can be used to calculate the average time it will take to identify the cube. The minimum time taken will be t and the average number of failures before the first success can be calculated using $(1 - p) / p$. The average time taken to produce a successful result will be:

$(1 - p) / p * t + t$ = Average Time

p = 0.787 and t =0.765

Therefore:
Average Time = (0.213 / 0.787) * 0.785 + 0.765
Average Time = 0.972 seconds

5.2 Colour Calculation and Cube Configuration

The getColour class calculates the the configuration of the cube based on images of the six sides of the cube. There are three parts of the class where the ability of a method should be tested, getting the colour of the nine squares of the cube in an image, ensuring that there are nine or each colour and finally calculating the configuration of the cube from the colours. The first test was performed by placing the cube in a fixed point in front of the camera, the system was told the position of the cube and had to calculate the colours of the squares. The test produced three results for each face of the cube and this was repeated fifty times on different faces of the cube. All of the colours of the cube were correctly identified in 126 of the 150 tests. It should be noted that tests would get the same results when they were initially repeated. Of the 24 remaining tests only one of the colours was incorrectly identified.

To test the ability of the system to correct the colours of the faces, images were taken of the six faces of the cube. The colours of the squares were then calculated for each of the six sides. The number of each of the six colours was counted, if there were nine of each colour the cube was assumed to be correctly identified. The system then selected one of the 54 squares at random and changed them to one of the other five colours at random. The test was also completed with two and three squares selected and changed. The *fixColours* method then tried to correct the colours of the image and the result was compared to the original value to calculate the ability of the method. This was repeated 1000 times on one cube and repeated for five different cubes. There were 15000 tests in total.

	Number of squares changed.		
	One	Two	Three
Unsuccessful	91	334	687
Failure Rate	1.80%	6.70%	13.70%

The negative results of these tests were further examined and it was found that if two squares swapped colour so that there will still nine of each colour then the system did not identify a problem and so did not correct it. (diagram) This would later be resolved when the class attempts to calculate the configuration of the cube in the *getCornersAndEdges* method and finds that it is not possible, the images of the cube would then be retaken.

The ability to calculate the configuration of the cube was difficult to test as the colours could not be positioned at random due to the unique pieces of the cube. The Rubik's cube is stored in the same manner for the *solver* class as the output from the *getCornersAndEdges* method and the *writecolour* method converts the configuration of the cube into the colours of the squares. To test the *getCornersAndEdges* method a cube was created by the *solver* class, it was then shuffle and the configuration noted. The *writecolour* method creates a string of colours which is formatted specifically for the *getCornersAndEdges* method. The *getCornersAndEdges* method then attempts to calculate the configuration and the result is compared to the original configuration. This was repeated 10000 times and was successful ever time. To check that the test was working correctly a single not central colour was change before the *getCornersAndEdges* method was run and the method failed 100 out of 100 times. The central colours were not changed as they would not affect the configuration of the cube.

The testing of the getColour class was difficult because in the final system the robot rotates the cube and the control class will take image of the six sides which are then possessed. At the time that the

getColour class was tested the robotic part of the system was not complete. The author had to manually manipulate the cube and ensure the position was correct when the images were taken and so some human error could have occurred.

The chance of success can be calculated by finding the chance of the colour of a square being incorrectly identified and calculating the chance that the *fixColours* method will fix the problem. As the *getCornersAndEdges* will always succeed this does not need to be factored into the equation. The probability *PXY* of a colour being incorrect identified exactly *X* times given the number of faces of the cube *Y* and given the chance of failure on one face being *P* can be calculated using the equation:

$$((Y ! / (Y - X) !) / X !) * (P)^X * (1 – P)^(Y - X) = PXY$$

The probability *P* of the failure on one face was calculated previously as 126/150 or 0.16.
The number of sides of the cube Y is always 6.
The X value will vary between 0 and 3 as these values were tested for the *fixColours* method and the chance of 4 failures or more is less than 1%.

The chance of X failures is calculated and multiplied by the values calculated by the tests done in the *fixColours* method Pfix these can be summed to find the overall chance of success.

Where X varies from (0 to 3)
PXY*Pfix ~ Overall chance of success

when X=0
(1 – 0.16) ^ 6 = PXY = 0.351
0.351*1 = 0.351 (chance of successful *fixColours* method is 1 if X is 0)

when X=1
6 * (0.16) *(1-0.16)^5 = PXY = 0.401
0.401*(1-0.018) = 0.394

when X=2
((6*5)/2) * (0.16)^2 * (1-0.16)^4 = PXY = 0.191
0.191*(1-0.067) = 0.178

when X=3
(120/6) * (0.16)^3 * (1-0.16)^3 = PXY = 0.049
0.049*(1-0.137) = 0.042

0.351+0.394+0.178+0.042 = 0.965 = chance of success

The equation used before to calculate the average time taken to successfully identify the Rubik's cube in the image can be used to calculated the average time to calculate the configuration of the cube. However for this example the time taken for a failed attempt *tf* is higher than the time taken for a successful attempt *ts*. The robotic hands must rotate the cube for images of each side to be taken. There are eleven rotations of the robotic hands in the process of taking the images and it takes approximately 0.9 seconds for a each rotation. It also requires eleven rotations to reset the cube and hands to the original positions if the class cannot correctly identify the cube. The equation to calculated the Average time to successfully identify the cube will be:

$(1 – P) / P * tf + ts$ = Average Time

ts = 11*0. 87 =9.57
tf = 22*0.87 = 19.4
P = 0.965

(1-0.965)/0.965*19.4+9.57 = Average Time = 10.27

5.3 Cube Solver

The *solver* class of the algorithm was tested by shuffling the cube, calculating the manipulations to return the cube to its solved state, performing the manipulations and then testing if the cube was solved. The cube was shuffled by selecting twenty five manipulations at random and performing them on the cube. The part that required testing was the *solve* and *optimise* methods which were run once the cube had been shuffled. This produced the series of manipulations that were required to solve the cube. These manipulations were then performed on the cube and a *checkSolved* method was used to check if the cube was in the solved state. A counter was incremented every time that the cube was not correctly solved. This was run 30,000 times and did not fail on one of the occasions. The ability to test the solver class was also tested by purposefully adding one incorrect manipulation to the cube before the *checkSolved* method was run and of the 1,000 times the test was run the *checkSolved* method correctly identified that the cube configuration was not solved.

The time taken to solve 10,000 Rubik's cube was recorded as 1.27*10 13 nanoseconds which is just over three and a half hours with an average time of 1.27 seconds per cube. The average number of manipulations was also recorded and the cube on average requires 77.6 manipulations to solve.

The cube solver took 77.6 manipulations on average to solver the Rubik's cube this value can be used by the adapter to robot to calculate the average number of robotic manipulations. The solver was expected to solve the Rubik's cube in far fewer manipulations than this, it is due to the one of the four parts of the Friedrich methods is performing poorly.

The Adapter to robot class would convert manipulations from the solver class into movement by the robotic arms. The class was tested by creating a virtual cube and testing the manipulations and the order that the manipulations are performed. The program randomly generated a series of 77 manipulations and then attempted to perform them. The class tracked the position of each of the faces whilst the cube was being manipulated, calculating when a manipulation were performed and the face that it was performed on. The manipulations which were actually performed were then compared with the manipulations which were randomly generated to determine the ability of the Adapter to robot class. The test was repeated 100000 times and it was found to be correct every time. The test was modified to check if both hands were simultaneously open or if both hands were close and in the vertical position. This was done to ensure that errors would not occur due the cube being dropped or the two hands colliding. The test was repeat, of the 100000 times no problems occur with the ability of the movement or the occurrence of errors.

When designing the system it was found that if a hand was rotated too far in one direction from the original position then cable to the hand would become too taught and the hand would no longer rotate .This occurred when the hand was rotated 720^{O}, however it is best to be cautious and so the system was tested at 540^{O} to reduce to lower risk of problems caused by tangled wires. The class was also tested to check how often the hand would rotate to 540^{O} or more in one direction. The hands would rotate to 540^{O} or more 4.5% of the time and so the robotic arms may not work

correctly. After this test it was decided that the code should be modified so that if the rotation in one direction was greater than 360^{O} then the hand would be rotated back 360^{O} to reduce the tension on the wire. The testing was then repeated and the hands were never rotated to 540^{O} or more from the origin.

5.4 Robotics

A test was going to be run for the time taken for the software to convert manipulations into movements however it was clear that the time would be negligible due to the time taken to do the tests 10000 taking approximately 3 seconds. Finally the number of moves, opening and rotating the hands, that are required to perform 77 manipulations was tested. When the test was run 100000 times it required 324 movements on average by the robot to perform manipulations. The figure of 324 movements can be multiplied by the time taken for a movement 0.87 seconds to estimate that the manipulation part of the system will take on average 281.9 seconds.

The implications of the results of the testing of the *AdapterToRobot* class are that it will always successfully convert a series of manipulations of the cube into a series of movements of the robot which will not overly rotate any of the wires and will not cause collisions between hands. This has no implications on the ability of the physical robot arms and is only testing the software part of the system.

The robotic part of the system failed to properly hold the Rubik's cube and so the cube could not be physically manipulated. Every programming part of the system that was required worked correctly however the engineering part of the project was unsuccessful. This meant that the system could not be tested as a whole. However the average time that it would take for the system to run could still be estimate.

	Hough Transform	Configuration Calculation	Cube Solver	Robotic manipulation	Total
Time Taken(s)	0.97	10.27	1.27	282	294.4

The time taken for the each of the sections of the system conforms to expectations as sections involving the rotation of the robotic arms take far longer. The total time take is most heavily influenced by the final section of the project which is the manipulating of the Rubik's cube to return it to a solved state. The Robotic manipulation time could be decrease by reducing the time taken to perform a manipulation of the cube, reducing the number of manipulations required to solve the cube and producing a more efficient way of adapting the manipulations of the cube to the rotations of the robotic arms.

The project generally was quite successful as all of the programming parts of the system were successfully implemented in the system. The approach to this project was suitable as the project was designed and implemented in iterations, which made the design more flexible. The programming language Java was used for the project and was appropriate the the was easy to access open source software available for both the web cam and the Lego Mindstorms.

6.0 Future Work

The final system did not function correctly due to the robotics arms being unable to deal with the weight of the cube. If further work was carried out on this system then improving the robotic arms would be a priority. The robotic arms were severely limited by the available construction materials, and a large amount of time was wasted designing the robot. The robotic arms could be improved

greatly by constructing them from stronger materials or improving the engineering ability of their designer.

Improvements to the cube solver were originally planned to occur once the image processing section of the system had been completed. However the time taken to implement the image processing part of the system took longer than expected as the reliability of the identification of the cube had to be improved. The cube solving method is divided into four separate parts and three of the four sections are efficient, however the second method *solveCorners* when tested was on average responsible for 54 of the 77 manipulations. This is the second highest priority if improvements are intended to be performed on the system.

The Hough Transform and Colour configuration sections of the system could be generally improved by increasing the chance of successfully identifying the cube or reducing the the time taken to identify the cube. However these sections have quite high chances of identifying the position or configuration of the cube and they take very little time, so this would have little effect on the system

Some additional features that could be added to the system in the future is the addition of one or more robotic hands to try to increase the speed of the system.

7.0 Conclusion

In conclusion the system was successful as every computing part of the system worked correctly and the aims of the these parts of the system have been met. The robotic part of the system does not solve the Rubik's cube which is a failure of that part of the system as it was key to the project. The parts of the project which do work, function quite well as the have low failure rates and ways to return to the original cube position if a problem occurs.

8.0 Reflection

Many things have been learnt from the creation of this project, this does not simply include improving my programming ability, but recognising the change in my ability to work on large projects over a long time scale. It is very difficult to manage the timing of the project as I can get stuck trying to improve a specific part of the system, before I more on to the next section. I should use Gantt charts and milestones to help decide how much time to spend on each part of the system. I often struggle to keep these charts updated and so there should be a regular schedule for updating any progress on the chart so that the project does not fall behind schedule. The time management skills learnt from this project will be transferable to almost all of the projects I will be a part of in the future.

One of my strengths in this project is my great interest in it, I really enjoyed implementing the solver of the Rubik's cube and the image processing classes as there was a large amount of problem solving involved. The solution to the problems in this system were often unclear, in the example of the Rubik's cube, it was very difficult to view the current state of the cube in the program. This made it difficult to test the system and a small piece of code which displayed the cube had to be written to show the current configuration of the cube. This was not an obvious way of resolving the problem and this may help me think of different ways of solving problems in the future.

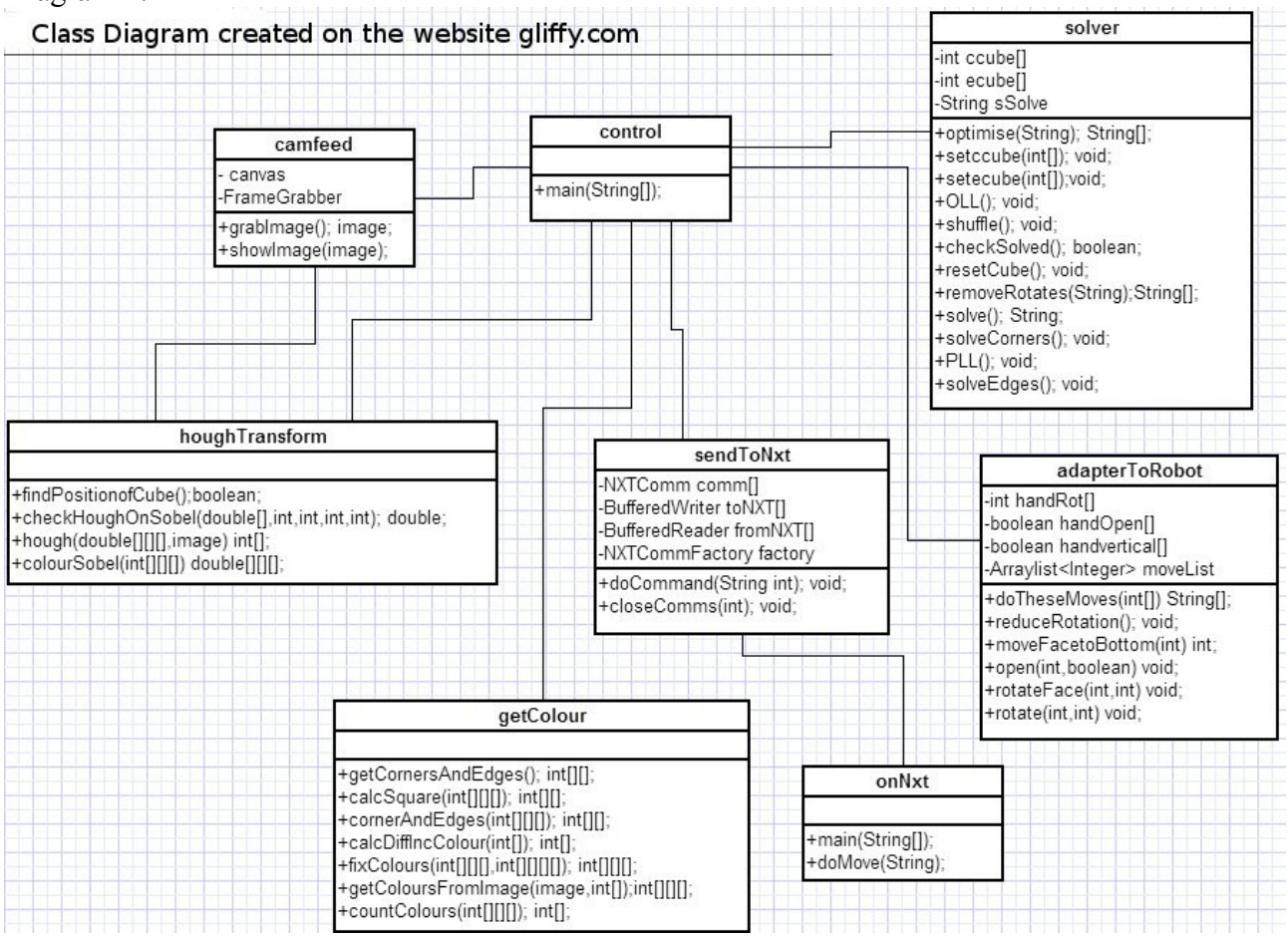
I was in communication with my supervisor every via email week and would occasionally ask him questions on certain aspects of the project. The meeting that were held every week helped me to keep the project from falling behind schedule and ideas on the projects could be expressed.

The parts of the project which I would have change would be the robotics part of the project I found it very time consuming and connecting the NXT bricks to the computer was annoying to implement as it was difficult test for errors. I am also unhappy that the robotic part of the system did not work as if felt it. Take attention away from the working parts of the system, this will also make it difficult to demonstration the working parts of the system.

Appendices

The Java code is contained in these files and can be found in the archive
adapterToRobot.java
camFeed.java
control.java
getColour.java
houghTransform.java
onNXT.java
sendToNXT.java
solver.java

Diagram 1.



References

1. LeJOS, Java for Lego Mindstorms. 2013. LeJOS, Java for Lego Mindstorms. [ONLINE]

Available at: http://lejos.sourceforge.net/. [Accessed 03 May 2013].

2. 2013. . [ONLINE] Available at: http://www.wdjoyner.org/papers/gods-number3.pdf. [Accessed 03 May 2013].

3. Image Transforms - Hough Transform. 2013. *Image Transforms - Hough Transform*. [ONLINE] Available at:http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm. [Accessed 03 May 2013].

4. Rosin, P, 2013. Lecture Slides. Image Processing

5. Unexpected leading characters in buffered reader (Beginning Java forum at JavaRanch). 2013. Unexpected leading characters in buffered reader (Beginning Java forum at JavaRanch). [ONLINE] Available at:http://www.coderanch.com/t/601782/java/java/Unexpected-leading-characters-buffered-reader. [Accessed 03 May 2013].

6. . 2013. . [ONLINE] Available at:http://homepages.inf.ed.ac.uk/rbf/IAPR/researchers/D2PAGES/TUTORIALS/VisMac2008.pdf. [Accessed 03 May 2013].

7. . 2013. . [ONLINE] Available at:http://www.gcu.ac.uk/media/gcalwebv2/theuniversity/academicschools/sls/psytimetables/LogvinenkoTokunagaAPP_OnlineFirst.pdf. [Accessed 03 May 2013].

8. Mark Nixon, 2012. *Feature Extraction & Image Processing for Computer Vision, Third Edition*. 3 Edition. Academic Press.