Interim Report
Will Pitt C1015111

Abstract
The overall aim of this project is to create a device which will solve a Rubik's cube by taking pictures of it to determine its state, and then manipulating it using “Lego Mindstorms”, whilst checking that the system is working correctly by processing images of the cube.

Table of Contents

Introduction

The overall aim of this project is to create a device which will solve a Rubik's cube by taking pictures of it to determine its state, and then manipulating it using “Lego Mindstorms”, whilst checking that the system is working correctly by processing images of the cube.

This process can be broken down into smaller stages which will be easier to manage, test and program. One part of the project will be taking images of all of the sides of the cube and then processing the images and calculating all of the positions of the pieces of the cube. The next parts of the program will need to solve the cube after receiving the data from the first part and then pass the method of manipulating it to the final stage of the process. The third part of the process will be to send the method of solving the cube to the Mindstorms' NXT 2.0 “brick”, the central processing unit of the Mindstorms system, which will be programmed to understand the commands and manipulate the cube so that it is solved. The project will be written in Java on both the NXT 2.0 and the computer.

The specific aim of the system is for the program to solve the scrabbled cube in less than 40 moves. I used the value of 40 as a target for my system as I have found that the number of moves that the cube solving robots in competitions take tend to be around 30 and so to compete with them I must have a system that solves the cube in a similar number of moves. The system needs to be able to recognise when the cube is not correctly positioned within the cradle of the Mindstorms assembly and should try to correct the cube in the most appropriate way possible. The system needs to optimise the way that the cube is solved based on the way that the mechanical part of the system acts as it will be easier for me to change the way that the program works rather than changing the mechanical part of the system. The software needs to identify the way that the cube has been scrambled by taking pictures of each of the sides of the cube and then needs to check if the cube has been correctly put together as it is possible to have an unsolvable cube.

This project and projects similar to this one have been done several times in the past by people who have spent several years researching the Rubik's cube and so it is unlikely that I will personally bring anything revolutionary to the field. There are also many projects and competitions between

universities to create the fastest machine to solve a Rubik's cube which is where I was influenced from to start this project. This has meant that my goals for the project are not to create something which has never been done before but more to find new methods of trying to solve a problem to aid in the field. These methods could then be used alongside other methods to find a better way of solving the problem. The beneficiaries of the project will be people who themselves are working on finding ways of solving Rubik's cubes, it is likely that there will also be people working on other solutions to logic puzzles that will find it useful to read some of the methods that I have used as they may be applicable to some of their own problems.

My approach to the project was to research methods for solving the Rubik's cube and try to understand what methods would be possible for me to implement given the time constraints and the amount of knowledge in that field that will be required. I researched the human methods that are used as well as the mathematical methods that are used to solve the cube and found the mathematical methods to be far more complicated. I felt that I should start by programming the human methods for solving the cube and then move on to using the mathematical methods when I had a better understanding of the field later on in the year. The human method that I am using is the “Friedrich Method” which involves several steps that can be split into four main processes (1).

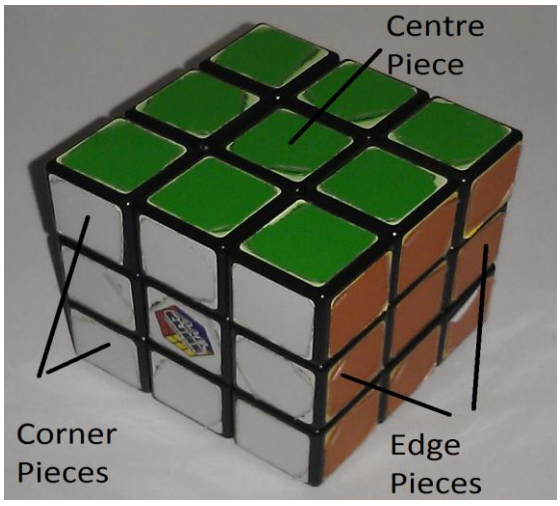
I created an initial plan for the project which outlined the aims, the objectives and a description a description of the intended finished project. I then used the aims, objectives and description, along with my research to estimate the achievable milestones of the project so that I had targets to aim for to ensure that my project was completed on time.

I am assuming that the processing power of my computer will be enough for the requirements of my program and that I may need to change the computational complexity of the program to ensure that the time taken to solve the cube is reasonable.

I hope that outcomes of this project will be new and useful methods for solving a Rubik's cube as well as a quick and optimised method which does not require much processing power. The speed of the mechanical part of the system is less important as it is not expected of me to produce an efficient mechanical device which is not relevant to my course. It is likely that I will use a design that I have found on the internet for the mechanical part of the system so that I do not waste time on designing the device. I will however be able to optimise the program so that it is suitable for the mechanical device used by the system.

The Background

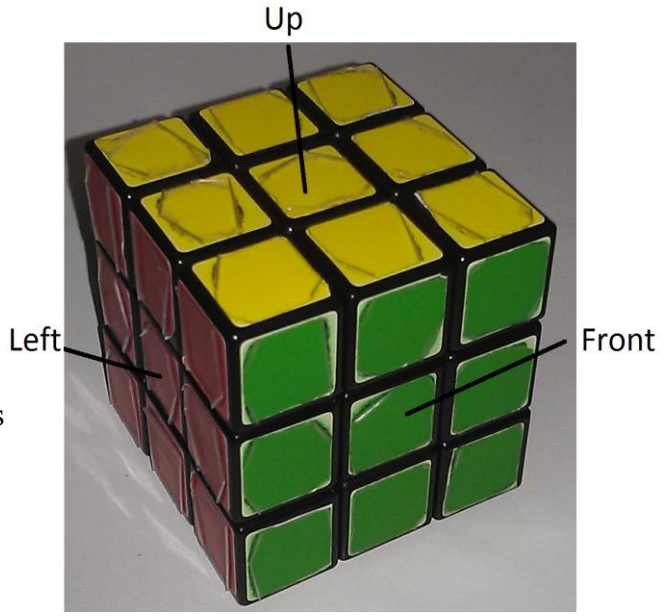
I will use Java for all of the software in the program as this is suited to the NXT 2.0. I will use Java to run the software on the computer for the camera and the solving of the cube to make it easy to communicate between the Lego Mindstorms and the computer and because of all the programming languages I am most knowledgeable at programming in Java.



In addition to the programming aspects of the system there will be many non-computing aspects. The Rubik's cube is a logic puzzle which is made up of 20 different pieces moving around a centre piece. The Rubik's cube

is a cube where each face appears to be made of 9 square pieces however the central pieces of each face cannot move and can only be rotated. The cube can be manipulated by rotating any of the six faces. These faces can be rotated to four different positions (including the original). This results in 18 different possible ways of manipulating the Rubik's cube. The cube can be thought of as 8 corner pieces which can each be orientated in three different directions and twelve edges pieces which and each be orientated in two different directions.(Fig.1)

There is a very large number of possible cube configurations if the Rubik's cube is completely disassembled and then reassembled at random, but only a small percentage of these configurations (roughly ten percent(2)) are valid in that they can be achieved by manipulating the cube. So for example the red centre piece will always be opposite the orange centre piece in a valid cube configuration. The initial configuration of the cube has to be produced by shuffling; this is achieved by rotating random faces of the cube for twenty five moves. Google have experimented on the maths behind the Rubik's cube and have recently proved that the maximum number of moves required to solve any cube is twenty(3). So the system should only need to rotate faces just over twenty times to achieve every valid cube configuration. The cube will be rotated for over twenty moves due to the random element of the program which means that the cube may undo some of the manipulations it has done.



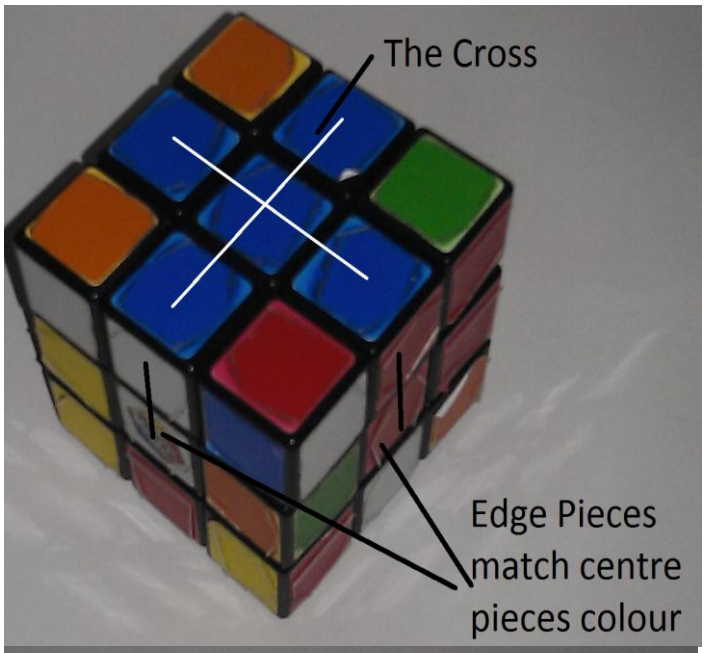
Wherever I refer to the cube I will act as if the cube is on a flat surface with one side facing the viewer. I will refer to each face by a given name, where these names are: front or “F” for the face which is facing the user; “B” or back for the face on the opposite side of the cube to the front face; “U” or up for the cube which is on the top and facing upwards; “D” or down for the face which is opposite the up face and touching the surface the cube is on and finally “L” and “R” or left and right for the two remaining faces on the sides of the cube (Fig.2). Each face can be rotated to three different new positions when manipulating the cube. These manipulations will be represented by the letter on its own for a clockwise 90° movement, for instance “B” or “F”. A number “2” will follow the letter if the cube face is moved 180°, for instance “L2” or “R2”. A single quotation mark will follow the letter if the cube is rotated 270° (i.e. 90° anti-clockwise), for instance “ U' ” or “ D' ”.

Mindstorms is a piece of technology which can be programmed to do various tasks and has a central processing unit known as the NXT 2.0 brick. The brick is designed to connect to Lego parts so that it can easily be incorporated in Lego models, and the

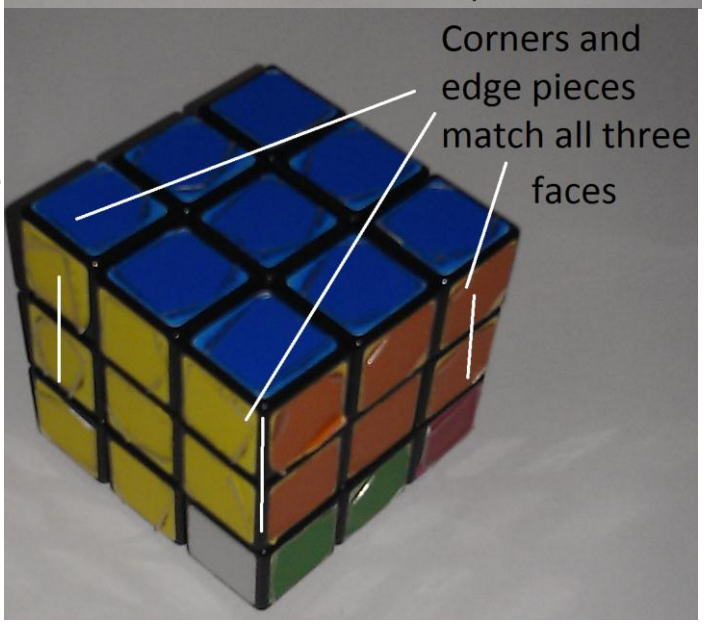
designer can then produce a vast number of different projects using the Mindstorms kit. It can receive inputs from various sensors that can be connected to the NXT 2.0 or receive data from a computer. The sensors can be Lego branded or third party, and the NXT 2.0 can then process the sensor data using the original Lego software or Java classes written by and uploaded by the user. The NXT 2.0 can also output data by sending messages back to the computer or driving motors and other output devices connected to the brick(Fig.3)(4).

The problem that the project aims to solve is that the current methods of solving a Rubik's cube take a large time to compute, but require few manipulations. Alternatively they don't take very long to compute but require a large number of manipulations. The objective of my project is to generate a different approach to solving the cube which will work well with the available mechanical part of the system.

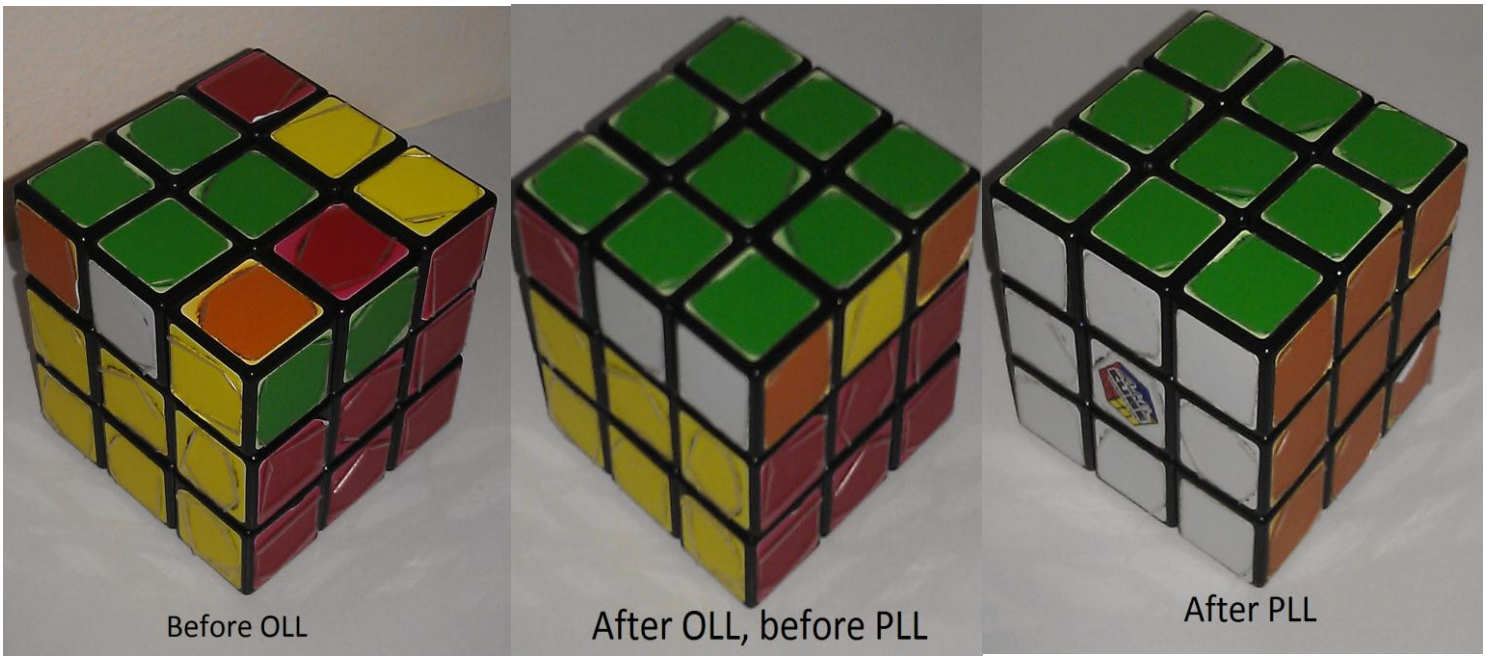
The theory behind solving a Rubik's cube using human based methods involves more viewing of the cube and deciding what to do without having to remember large amounts of information about algorithms to solve the cube or using complicated maths to calculate the best way to solve the cube. This means that the human methods are much easier to understand and implement (by humans), but are not as fast as the maths based methods for solving the cube. One of the more advanced human based methods of solving the cube, known as the “Friedrich method”, involves splitting the solution of the cube into four steps. This is the method that I will use for my cube solver before moving onto the maths based methods(1).



To use the Friedrich method the program must first solve a cross on any side of the cube. This is where the four edge pieces around a centre piece match the colour of that piece and also match with all the centre pieces of the four faces that touch that side. (Fig.4) The next step in the Friedrich method is to fit the corners of the cube that are beside the cross as well as the edge pieces next to the cube beside the cross so that the bottom two thirds of the cube are solved.(Fig.5) The next step is to orientate the top face of the cube so that all of the pieces of the top face have the correct face pointing upwards this is know as Orientation of the Last Layer or OLL. There are fifty seven different ways that the eight pieces can be orientated and an algorithm to solve each of the orientations.



(Fig.6) The final step is to swap the pieces of the cube on the top face until all of the edges around them match the faces beside them. This is known as Permutation of the Last Layer or PLL. (Fig.7) There are twenty one possible ways that the final layer of the cube can be arranged and an algorithm for each one.(5)(Fig.8)



My research on mathematical methods of solving the Rubik's is based on the research paper “Algorithms for Solving Rubik's Cubes”(6). I have found that the methods used in this research paper involve grouping the Rubik's cube pieces (which the authors refer to as “cubies”) based on the permutations of their positions due to the limited possible moves of a Rubik's cube. The paper shows research on n*n*n cubes rather than just simple 3*3*3 Rubik's Cube and explains how the cubies should be grouped into clusters, where a cluster is all the possible positions a cubie can be moved to by manipulating the cube. Each cluster can be solved separately from all other clusters without affecting the positions of any of the other cubies outside that cluster. An edge cluster is any cluster containing a cubie with two faces on and all cubies in that cluster will be edge cubies. A corner cluster is any cluster containing a cubie with three faces on and, as with edge clusters, all cubies in that cluster will be corner cubies. The two clusters of a 3*3*3 Rubik's cube are all of the edges as one cluster and all of the corners as another cluster. The way that this research is related to my problem is that the aim of using this method is to combine the cluster move sequences of the two clusters of a Rubik's cube to construct a sequence of moves which can be resolved concurrently. This research could also be useful if there is further work on cubes where n is greater than 3(6).

Due to the mechanical device that I am using being constructed out of a plastic construction toy, the speed of the manipulation of the cube is likely to be slower that a custom made metal robot. These limitations will be due to constraints in the proximity of components and the strength of the motor available for Mindstorms. The cube that I am using for the system also has some limitations as it has been used many times and so colours of the squares of the cubes are harder to distinguish and this should be taken into account when processing the images of the cube. The system performance will also be affected by lighting conditions.

Existing solutions relevant to the problem area include several devices which solve a Rubik's cube, some of which are made out of Lego Mindstorms and some that are made from metal, custom made for Rubik's cube solving. The most notable cube solving machine is the "CubeStormer II" which current holds the World record for the fastest solution of a Rubik's cube(7).

Several other people have already made devices similar to the one that I hope to create, however I am going to take ideas from other projects for the mechanical parts of the system, but will provide my own programming solution. There are many different methods for solving the Rubik's cube and so my approach should be a different method to most of the methods currently being used. I can use these projects to support my system by looking at where they may be slower than other projects or may fail in certain areas and try to improve my project in the areas that the other projects are better than mine.

There was a very helpful website(5) that I used which showed how to identify each of the orientations and permutations, and the algorithms that were involved with resolving them. However the website only showed how the cube looked for each of the orientations and permutations. I had to translate them into a format that was readable by my program. This involved calculating the orientation or permutation for each piece and converting it to an integer for all fifty seven orientations and twenty one permutations. The algorithms that were present on the website for completing the OLL and PLL parts of the solution were in a form which was not usable by my program as it involved non standard manipulations as well as rotations, whilst the cube was being manipulated. I had to modify most of the algorithms so that they were appropriate for my system and they can be found in the "OLLA.txt" and "PLLA.txt" files attached to this report.

The Specification and Design

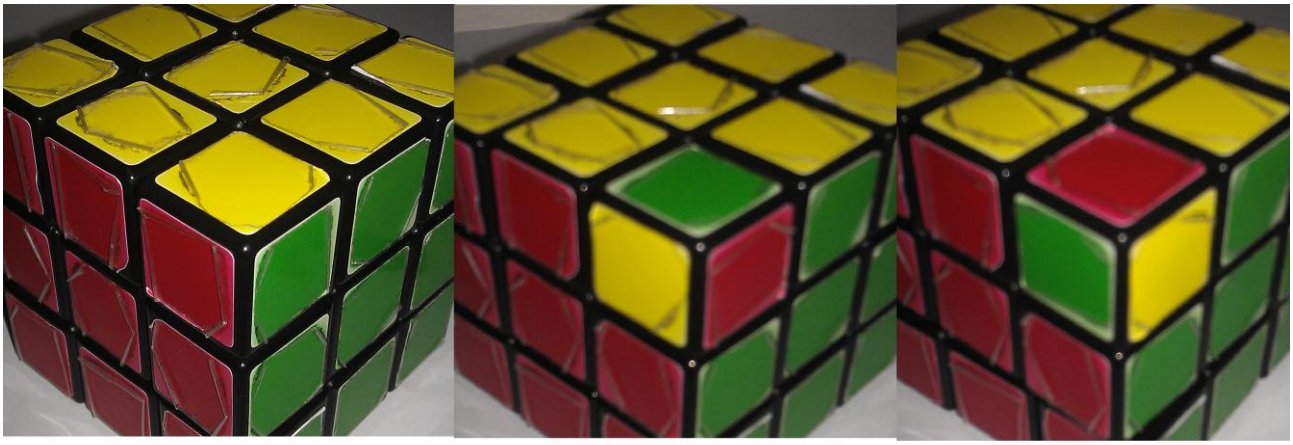
The system consists of three primary classes which are linked together using a *main* class which controls all of them. The *camera* class is one which will take pictures of the Rubik's cube and then return these pictures in a format which the other classes will be able to interpret. The *main* class controls all of the other classes and must decide how to deal with problems in the system if they occur. The *image processing* class will analyse and find the colours at each of the squares of the three by three face from the picture given and then calculate and return the positions of the pieces of the Rubik's cube. The *status* class will also look at an image of the cube and decide whether the cube is correctly orientated and whether there is an error to report. The *solving* class will be given the positions of the cube and then solve the cube and return an array of the moves to solve the cube. The *Mindstorms* class will run on the NXT 2.0 and will interpret and execute all of the moves that it is instructed to do. It will also tell the *main* class if a button has been pressed that is connected to the brick. Refer to diagram at the end of the document for further detail.(Fig12)

The data must flow through the system in a way that ensures that the current state of the cube is up to date and there is no backlog of data in the system. The *main* class will wait for an input from the NXT 2.0 to say that the button has been pressed to start the system. The *main* class will tell the *camera* class to take pictures whilst telling the NXT 2.0 to rotate the cube to get images of all the sides. The *camera* class controls the camera which is physically connected to the computer via USB. The *main* class will then send these pictures to the *image processing* class of the system which will calculate the cube and then send an array back to the main program of all of the positions of the pieces of the cube. The *main* class then sends this data to the *solving* class, which will solve the cube and output an array of all of the manipulations that are used to solve the Rubik's cube. The *main* class will then work out how these manipulations should be carried out in the

Mindstorms class and sends the manipulations to the NXT 2.0 which will physically manipulate the cube. While the *Mindstorms* class is manipulating the cube, the *main* class will run the *camera* class to take pictures at regular intervals to check the status of the cube. These images will be sent back to the *main* class and the images will be sent to the *status* class to ensure that the cube is still correctly positioned. If the cube does become incorrectly positioned then the *main* class will be told by the *status* class and can try to get the *Mindstorms* class to correctly reposition the cube, and if it is successful it will start the process from the beginning by re-examining the cube with the *camera* class.

The data types that are used in the system are image files which are sent from the *camera* class to the *main* class and then to the *image processing* class. An array of moves are sent between the *Mindstorms* class and the *main* class. These must be sent in packets by wiring the computer to the NXT Brick by USB or via Blue-tooth. The *main* class will call the other classes to run them and they will all return standard Java arrays so that data can be passed easily around the system.

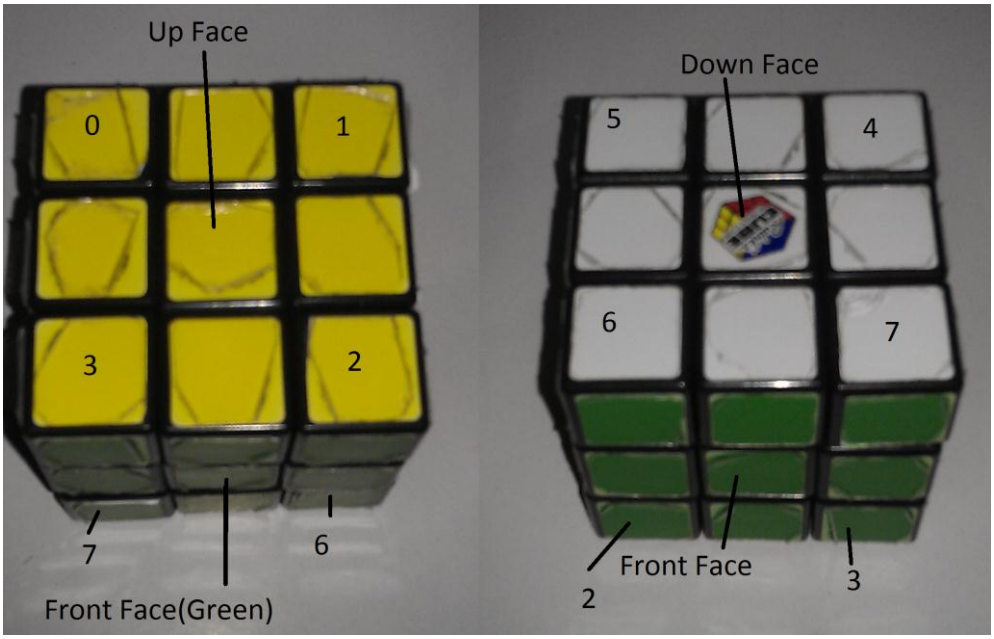
The algorithms involved with the *camera* class will be similar to a design pattern known as an adaptor.(8) When the *main* class runs the *camera* class, it will use a method to take a picture using the camera attached to the computer, then use another method to turn the data from the image into a very simple image file so that it is easily readable by the other classes. The *image processing* class will be given all six images of the faces of the cube and then it will run a method that uses the images to find the average brightness of the cube. This value is then sent to a method which will return a three by three array of all of the colours of the face of the cube in the image. This will be calculated by placing a grid over the image of where the nine squares of the Rubik's cube are expected to be and then comparing the value of the average colour, with respect to the average brightness, in the area specified by the grid. The *status* class will be given an image from the *main* class and then check that the cube is correctly positioned and give the type of error that has occurred to the *main* class.



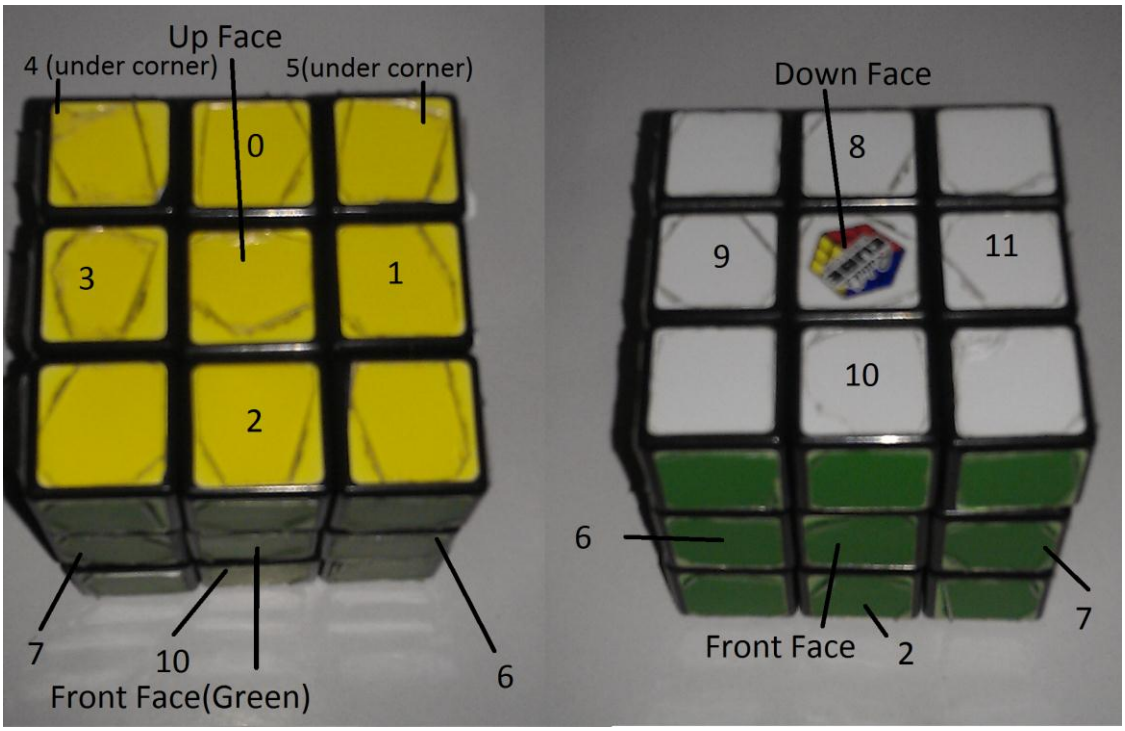
Orientation = 0 Orientation = 1 Orientation = 2

The *solving* class will have a global cube which can be modified by any manipulation method and any rotation of the cube between manipulations. The way that the cube is represented is very important as it should be easy to modify with the manipulation and rotation methods as well as being easy to check and compare when the PLL and OLL variation is calculated. I decided to represent the cube as two arrays one for the corners and one for the edges. The corners of the cube are represented in the program by an integer array, where the number refers to the type and orientation of the piece and the position in the array refers to its position in the cube. The cube piece

type is equal to the value in the array divided by three, when the pieces are correctly positioned if all of the piece types are equal to there positions in the array. The cube piece orientation is equal to the value in the array modulus three. The orientation value is 0 if the cube piece is correctly orientated, the value is 1 if the cube is rotated anti-clockwise once and 2 if it is rotated clockwise once. (Fig.9)(Fig.10)



Due to the cube having no clear way to reference the way that it was rotated, I always had the cube with the yellow face on the “up” side; green face on the “front” side; red face on the “left” side; blue face on the “back” side; orange on the “right” side and white on the “down” side. The cube could be placed in any orientation and still be calculated in the same way, however this will make it easier to explain the positions and orientations of the pieces. The cube piece types were between 0 and 7 and started in the top left hand corner when looking down on the cube, first the “up” layer and then the “down” layer. The way the edges of the cube were represented in the program was very



similar to the way the corners were with twelve integers in the array and the edge piece type is the value divided by two, the orientation is the value in the array modulus 2. The edge piece types when complete are stored in the array so that, starting at 0 and counting through, all twelve edge pieces are represented in the physical world by counting clockwise from the top, starting on the "up" layer, then the middle layer starting from the top left corner, and then the "down" layer in the same manner as the "up" layer.(Fig.11)

This method may seem complicated, however it made it easier to manipulate and rotate the cube. Calculating the OLL and PLL variations of the cube was much simpler than when I tried to represent the cube by the colours on each of the faces. I originally tried to represent the cube by the colours of the face but found that every time that I tried to check the cube, I would have to re-calculate which part of the cube I was looking at and it made the process much slower. I am going to re-factor the code used in this part of the system next semester so, that the Rubik's cube is represented as a class with separate integers for both the orientation and position of each piece.

Methods are used in the class for each of the eighteen ways of manipulating the cube, such as *L(), R2()* and *U'()*. There are also two rotation moves which do not manipulate the cube but rotate the entire cube. They are called *rotFL()* and *rotFD(),* where these stand for rotate "Front" face to the "Left" face and rotate "Front" face to the "Down" face. A third rotation is never required in this system (it can be achieved by combinations of the other two) and so only two were implemented. The first step of the Friedrich method is to calculate the cross, which is done in my program by brute force. The method tries every manipulation for up to eight moves until a cross is solved on any one side. As there are eighteen different moves this method can be quite slow if seven or eight manipulations are required and will take approximately five minutes to run if this occurs. However there is a very small chance of this happening. I tested this program to find the average time to do eight manipulations which was approximately five minutes; I discovered this by using the brute force method and running this method but never checking for a cross on the cube. The next step of solving the cube is to fit the corners and edges around the cube. This is done in two separate stages one part identifies the corner pieces and moves them into position; the second part identifies the edge pieces and moves them into position. These methods can be improved by combining the corner piece and edge piece steps into one step.

The next two methods used were solving the OLL and PLL which involved reading text files into the program, comparing them with the orientation values in the cube, and then executing the appropriate algorithm in the corresponding text file. The "OLL.txt" file is used to identify the orientation type by comparing a series of integers on a given line and rotating the cube until one of the orientations from the text file matches. The algorithm from the same line in the corresponding text file is then used to solve the OLL ("OLLA.txt"). A very similar method is used for solving the PLL of the cube, where the permutation of the final layer is compared with "PLL.txt" and then an algorithm is selected from "PLLA.txt" .

The *Mindstorms* class will have methods to receive messages from the *main* class and interpret these as commands. Another method must then run the motors of the Lego Mindstorms to manipulate the cube to the correct position.

If the system follows to plan it will not have a graphical user interface as the user will simply place the cube in the cradle of the device and press a button. Then the device will solve the cube and

allow the user to pick it up, although whilst I am testing the Rubik's cube I have written some software to display the state of the Rubik's cube.

Conclusion

The main findings of the research so far is that I know how to implement a large percentage of the classes of the Java program such as the Rubik's cube solving class and the *image processing* class. The only part of the system that is currently lacking in progress is the mathematical method for solving the Rubik's cube. The future work which needs to be completed for the final report includes: implementing the *Mindstorms* and *image Processing* parts of the system; improving the speed of the cube solving algorithm by changing the second step of the program into one single step; further research and implementation of mathematical methods for solving the Rubik's cube. I feel that the way that the project is progressing is meeting the time plan, and other than missing the milestone for completion of the *image processing* module, the plan has so far been met and so I do not feel that I need to change it.

Glossary

centre piece : a Rubik's cube piece which is only on one face
corner piece : a Rubik's cube piece with three faces on it
cubie : a Rubik's cube piece
edge piece : a Rubik's cube piece with two faces on it
face : one of the sides of a Rubik's cube
NXT 2.0 Brick : the central processing unit of the Lego Mindstorms kit
valid cube : a cube which can be manipulated to create a solved cube without disassembly

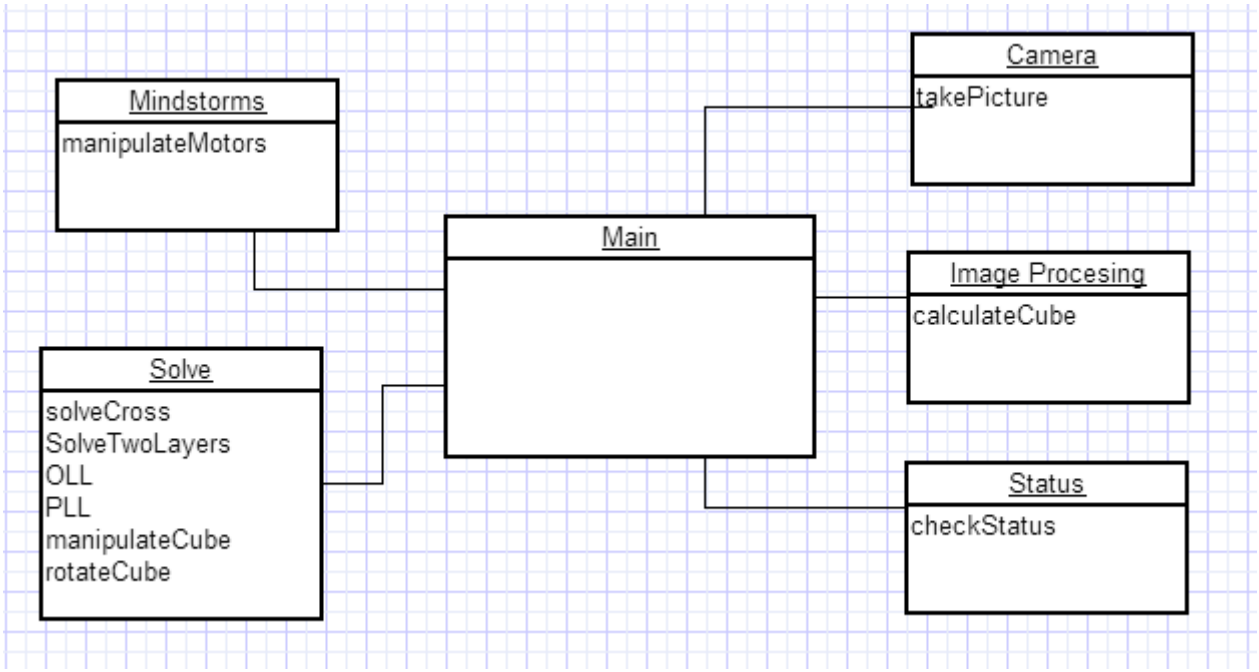
Table of Abbreviations

OLL : Orientation of Last Layer
PLL : Permutation of Last Layer

Appendices

colours.txt
display.java
OLL.txt
OLLA.txt
out.txt
PLL.txt
PLLA.txt

rrubiks.java
rubik.java



(Fig.12)(9)

References

1. http://www.cubewhiz.com/cfop.php
2. http://faculty.mc3.edu/cvaughen/rubikscube/cube_counting.ppt Counting the Permutations of the Rubik's Cube, Scott Vaughen.
3. http://www.engadget.com/2010/08/09/rubiks-cube-solved-in-twenty-moves-35-years-of-cpu-time/
4. http://mindstorms.lego.com/en-us/Default.aspx
5. http://www.cubewhiz.com/
6. http://arxiv.org/pdf/1106.5736v1.pdf (Algorithms for Solving Rubik's Cubes)
7. http://www.wired.co.uk/news/archive/2011-11/11/cubestormer-ii-world-record
8. http://www.oodesign.com/adapter-pattern.html
9. http://www.gliffy.com/gliffy/#