PES

PES Institute of Technology

[An Autonomous institution under Visvesvaraya Technological University]

100 feet Ring Road, Banashankari III Stage,
Bangalore-560085.

A PROJECT REPORT

On

"RUBIK'S CUBE PROBLEM SOLVER"

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF ENGINEERING

In

INFORMATION SCIENCE AND ENGINEERING

By

SINDHURI K . N	USN: 1PI08IS097
VARSHA ABHINANDAN	USN: 1PI08IS117
VEDASHRUTI PANDIYAN	USN: 1PI09IS119

Under the guidance of
Mr.Ashoka.M
Dept. of IS&E
PESIT, Bangalore-85

DEC 2011

Department of Information Science and Engineering
PES INSTITUTE OF TECHNOLOGY

PES

PES Institute of Technology

Bangalore-560085.

Department of Information Science and Engineering

CERTIFICATE

Certified that the project work entitled ***Rubik's Cube Problem Solver*** *carried out by* ***Ms. Sindhuri K N*** *USN* ***1PI08IS097****,* ***Ms. Varsha Abhinandan*** *USN* ***1PI08IS117*** ***Ms. Vedashruti Pandiyan*** *USN* ***1PI08IS119*** *bonafide students of* ***7th semester*** *in partial fulfillment for the award of* ***Bachelor of Engineering*** *in* ***Information Science and Engineering*** *of the PES Institute of technology, Autonomous College Under Visvesvaraya Technological University, Belgaum, during the year* ***2011*** *. It is certified that all corrections suggestions indicated for Internal Assessment have been incorporated in the Report. The project report has been approved as it satisfies the academic requirements in respect of Project course Work prescribed for the said Degree.*

Signature of guide

Mr.Ashoka . M
Assistant Professor
IS&E., PESIT

Signature of HOD

Prof.Shylaja S.S
Professor
IS&E., PESIT

Name of the student : Varsha Abhinandan
USN : 1PI08IS117

External Viva

Name of the Examiners | Signature and Date

1.

2.

ABSTRACT

Puzzles are great brain teasers and one such classic puzzle is the Rubik's cube. The Rubik's cube is a 3-D mechanical puzzle in which each of the six faces is covered by nine stickers, among six solid colours. For the puzzle to be solved, each face must be a solid colour. In today's world, technology has evolved to such an extent that humans want machines to do everything that had to be done manually earlier. We have gone a step further by using technology to understand and interpret difficult concepts, solve difficult problems as well and also to humanize the machine. Here is where the idea of a Rubik's Cube Problem Solver took shape.

This project tackles the problem of object detection and localization. More specifically, we attempt to locate a 3x3x3 Rubik's Cube by capturing a stream of pictures of each phase and extract the sticker colors using Computer Vision techniques with minimal help from the user. The application then shows steps to solve the Rubik's cube . We attempt to provide a solution to the problem by employing Computer Vision so as to make the interaction of the user with the Machine more natural.

Each face of the cube is scanned through a webcam. Edge and contour detection, square identification and colour identification algorithms have been used to detect the cube and its current configuration. Further, a layered approach has been followed to show the steps required to solve the cube. The output is a graphical result , favouring the user to understand the move sequence easily .

ACKNOWLEDGEMENT

Gratitude takes three forms-"*A feeling from heart, an expression in words and a giving in return*". The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of people whose ceaseless cooperation made it possible, whose constant guidance and encouragement crown all efforts with success. A Project of this kind can never be a success with the full fledged support of a group of reliable sources. I take this opportunity to express my heart-felt feelings.

Firstly, I would like to thank **Prof. D. Jawahar, CEO, PES Institutions** and **Dr. K N Balasubramanyam Murthy, Principal and Director of PESIT** for giving this opportunity.

Also, I would like to thank **Prof. Shylaja S S, H.O.D, Department of Information Science** for her guidance and allowing us flexibility in bringing our laptops to college.

I also thank **Mr. Ashoka . M , Assistant Professor, Department of Information Science** for her constant support and immense help in successful completion of the project. We also take this opportunity to thank all our faculty members for their helpful insights from time to time.

None on this world would have been possible without my parents: their encouragement assisted me to do this work; my heartfelt thanks to them.

Finally, I would like to thank my friends and all those people who have directly or indirectly helped me successfully complete this project.

VARSHA ABHINANDAN

Table of Contents

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Rubik's cube is a 3D mechanical puzzle invented by Hungarian sculptor and professor of architecture Ernő Rubik. Originally called the "Magic Cube"[8] the puzzle was licensed by Rubik to be sold by Ideal Toy Corp. in 1980[8] . It is widely considered to be the world's best-selling toy.[8]

In a classic Rubik's Cube, each of the six faces is covered by nine stickers, each of one of six solid colours, (traditionally white, red, blue, orange, green, and yellow). A pivot mechanism enables each face to turn independently, thus mixing up the colours. For the puzzle to be solved, each face must be returned to consisting of one colour.

A standard Rubik's cube measures 5.7 cm (approximately 2¼ inches) on each side. The puzzle consists of twenty-six unique miniature cubes, also called "cubies" or "cubelets". Each of these includes a concealed inward extension that interlocks with the other cubes, while permitting them to move to different locations. However, the centre cube of each of the six faces is merely a single square facade; all six are affixed to the core mechanism. These provide structure for the other pieces to fit into and rotate around. So there are twenty-one pieces: a single core piece consisting of three intersecting axes holding the six centre squares in place but letting them rotate, and twenty smaller plastic pieces which fit into it to form the assembled puzzle.

There exists several algorithm to solve the cube, but every algorithm has a disadvantage where in the cubist has to remember or memorize the complete steps of solving the cube. Later, there were a few applications which were favoured as they provided means of manually conveying to the system, the description of the scrambled cube and obtaining the specific steps to solve it. The manual entry of the cube's state was not found to be user friendly. Hence, the idea of video streaming of the cube's state and display of the steps to solve it arose.

1.1 Objective of the Project:

In this project we design and implement an application concerned with object detection and localization . More specifically, we attempt to locate a 3x3x3 Rubik's Cube by capturing the image of each of its face through a webcam and extract the sticker colors using Computer Vision techniques with minimal help from the user , so as to make the interaction of the user with the machine more natural. The application then attempts to provide a solution to the problem by applying Rubik's cube solving algorithms and displays the steps to the user in animated format for better understanding of the user.

1.2 Overview of Rubik's Cube Solving:

In Rubik's cubists' parlance, a memorised sequence of moves that has a desired effect on the cube is called an algorithm. This terminology is derived from the mathematical use of algorithm, meaning a list of well-defined instructions for performing a task from a given initial state, through well-defined successive states, to a desired end-state. Each method of solving the Rubik's Cube employs its own set of algorithms, together with descriptions of what the effect of the algorithm is, and when it can be used to bring the cube closer to being solved.

Most algorithms are designed to transform only a small part of the cube without scrambling other parts that have already been solved, so that they can be applied repeatedly to different parts of the cube until the whole is solved. For example, there are well-known algorithms for cycling three corners without changing the rest of the puzzle, or flipping the orientation of a pair of edges while leaving the others intact.

Some algorithms have a certain desired effect on the cube (for example, swapping two corners) but may also have the side-effect of changing other parts of the cube (such as permuting some edges). Such algorithms are often simpler than the ones without side-effects, and are employed early on in the solution when most of the puzzle has not yet been solved and the side-effects are not important. Towards the end of the solution, the more specific (and usually more complicated) algorithms are used instead, to prevent scrambling parts of the puzzle that have already been solved. In any case, remembering the complete steps and solving is a difficult task. Hence, the idea of creating applications was to help solving the cube easily for competing cubists as well as to help the beginners learn it .

1.3 Existing System and Problems:

Existing Rubik's cube solving applications are mainly of two kinds. The first kind requires to the user to manually enter the sticker colours of each face of the cube. The application then shows the user how the cube can be solved either in textual or graphical form. The second kind tutors the user on various cube configurations and what steps to follow from there to achieve a more solved stage. It then generates random scrambled cube configurations and presents it to the user to try and solve.

It is very obvious that the above mentioned approaches are time consuming for the user, not to mention the chances of mistakes in user input are very high, which tends to discourage users from using the application or sometimes even attempting to solve the cube altogether! Ease of use is a very important aspect of any application and these seem to lacking in that.

The existing application, favouring the purpose of solving the 3x3x3 Rubik's cube, illustrates many of the knowledge engineering problems encountered in building expert systems, such as knowledge representation, state space manipulation and the codification of expertise. Performance - that is, speed and efficiency - is a key issue and affects most of the design decisions in the program.

The system is a classic example of a "dumb" expert system. It unscrambles cubes as fast or faster than a human expert. However, it does not have the intelligence to discover how to solve Rubik's cube. It simply lists, in a form that the machine can follow, a collection of the rules that an expert would use to solve the cube.

Because of the large knowledge base involved in a system like this, a blind search strategy that picks rules at random simply will not work. For this reason, such system must use heuristic programming to direct the watch. Using various heuristic (intelligent) rules, the search space can be drastically reduced so that the problem can be solved in a reasonable amount of time.

This problem, like many other AI problems, resolves itself into three main design decisions. First, we must decide how to represent the current state of the cube to the system to analyse the state. Then, the system decides how to represent the things that can be done to the cube; that is, it conveys the steps followed to solve the cube. Finally, the cubist has the solved cube within seconds of time. Which is the design followed and implemented in our project.

1.4 Proposed System and Advantages:

The foremost goal of this project is to reduce the amount of interaction of the user with the machine. Hence, Computer Vision techniques have been made use of to let the computer "see" the 3x3x3 Rubik's cube itself and understand the cube's configuration. The input sensor used is a webcam through which picture frames of all the six faces are captured. Frames corresponding to the images of the six sides of the cube are collected and the sticker colours on each face are identified to build the configuration of the cube. This constitutes the first part of the system. The application of Rubik's cube solving algorithms to generate the solution steps constitutes the second part. These steps are displayed to the user in an animated form for the easier understanding of the user.

Object detection at first starts off with screening the image of the cube for obtaining an efficient image , to work along there are few necessary technical process related to the detection task to be adopted . The first jargon among those is Canny edge detection, which obtains all set of all edges in the captured video frame. This should be followed by contour identification, square identification and colour detection processes to obtain the necessary information from the image for the application of the cube solving algorithms. This information obtained is used to populate the data structure representing the cube and store its configuration.

The algorithm adopted for solving the Rubik's cube uses layered approach. Here the cube is considered as being made up of 3 layers. The edge and centre cubies of the first and second layer are first correctly positioned and oriented. Then, the corner cubies of the first layer are positioned and oriented. Now, the first layer is solved. Then, the edge cubies of the second layer are positioned and oriented to solve the second layer. Finally, positioning and orienting the edge and corner cubies of the third layer solves the cube.

CHAPTER 2

LITERATURE SURVEY

2.1 Computer Vision:

Computer vision is the field concerned with the automated processing of images from the real world to extract and interpret information on a real time basis. The goal of computer vision is to process images acquired with cameras in order to produce a representation of objects in the world to solve some task, or perhaps "understand" the scene in either a broad or limited sense.

There already exists a number of working systems that perform parts of this task in specialized domains. However, the generic "Vision Problem" is far from being solved. No existing system can come close to emulating the capabilities of a human. Vision is therefore one of the problems of computer science most worthy of investigation because we know that it can be solved, yet we do not know how to solve it well. In fact, to solve the "general vision problem" we will have to come up with answers to deep and fundamental questions about representation and computation at the core of human intelligence.

As a scientific discipline, computer vision is concerned with the theory behind artificial systems that extract information from images. The image data can take many forms, such as video sequences, views from multiple cameras, or multi-dimensional data from a medical scanner.

As a technological discipline, computer vision seeks to apply its theories and models to the construction of computer vision systems. Examples of applications of computer vision include systems for are controlling processes, navigation, detecting events, organizing information (indexing databases of images), modeling objects and environments, interaction (as input to a device for computer-human interface), automatic inspection, etc. Sub-domains of computer vision include scene reconstruction, event detection, video tracking, object recognition, learning, indexing, motion estimation, and image restoration. There are a lot of fields related to computer vision like artificial intelligence, optics and solid state physics, neurobiology, signal processing and computer graphics.

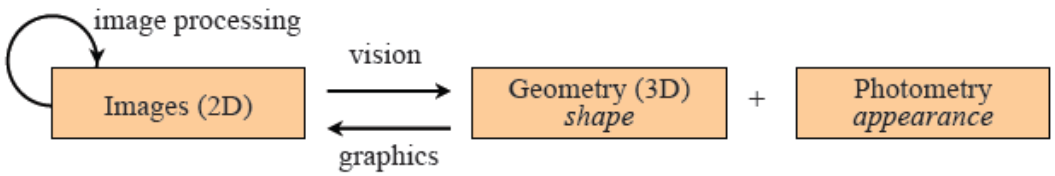

Figure 2.1: Computer vision in image processing

Why is vision so difficult? In part, it is because vision is an inverse problem, in which we seek to recover some unknowns given insufficient information to fully specify the solution. Some of the important issues in computer vision[2] are:

- **Noise –** Noise and distortions are caused by variations in the world (weather, lightning, reflections, movements), imperfections in the lens and mechanical setup, finite integration time on sensor (motion blur), electrical noise in the sensor or other electronics, and compression artefacts after image capture. Noise is typically dealt with by using statistical methods.

- **Segmentation** - In computer vision, segmentation refers to the process of partitioning a digital image into multiple segments (sets of pixels, also known as superpixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain visual characteristics. Thresholding, clustering, compression-based methods, and histogram-based methods are some of the techniques used for segmentation. Histogram-based methods are the most efficient image segmentation method and typically take just one pass through the pixels. In this technique, a histogram is computed from all of the pixels in the image, and the peaks and valleys in the histogram are used to locate the clusters in the image. Colour or intensity can be used as the measure.

- **Object detection**– If we are given an image to analyse, we would try to apply a recognition algorithm to every sub-window in this image. Such algorithms are likely to be both slow and error prone. Instead, it is more effective to construct special purpose

detectors whose job is to rapidly find likely regions where particular objects might occur. How the input, output and the intermediate information are represented and which algorithms are used to calculate the desired result are also important factors to be considered.

Computer vision is a rapidly growing field, partly as a result of both cheaper and more capable cameras, partly because of affordable processing power, and partly because vision algorithms are starting to mature.

2.2 Histograms and Matching:

In the course of analyzing images, objects and video information we frequently represent what we are looking at as a **histogram**. Histograms[6] can be used to represent such diverse things as the colour distribution of an object, an edge gradient template of an object, and the distribution of probabilities representing our current hypothesis about an object's location.

Histograms find uses in many computer vision applications. Histograms are used to detect scene transitions in videos by marking when the edge and colour statistics markedly change from frame to frame. They are used to identify interest points in images by assigning each interest point a "tag" consisting of histograms of nearby features. Histograms of edges, colours, corners, and so on form a general feature type that is passed to classifiers for object recognition. Sequences of colour or edge histograms are used to identify whether videos have been copied on the web, and the list goes on. Histograms are one of the classic tools of computer vision.

Histograms are simply collected *counts* of the underlying data organized into a set of predefined *bins*. They can be populated by counts of features computed from the data, such as gradient magnitudes and directions, colour, or just about any other characteristic. In any case, they are used to obtain a statistical picture of the underlying distribution of data. The histogram usually has fewer dimensions than the source data.

2.3 Thresholding:

In many vision applications, it is useful to be able to separate out the regions of the image corresponding to objects in which we are interested, from the regions of the image that correspond to background. Thresholding[11] often provides an easy and convenient way to perform this segmentation on the basis of the different intensities or colours in the foreground and background regions of an image. The input to a thresholding operation is typically a grayscale or colour image. In the simplest implementation, the output is a binary image representing the segmentation. Black pixels correspond to background and white pixels correspond to foreground (or vice-versa). In simple implementations, the segmentation is determined by a single parameter known as the intensity threshold. In a single pass, each pixel in the image is compared with this threshold. If the pixel's intensity is higher than the threshold, the pixel is set to, say, white in the output. If it is less than the threshold, it is set to black.

In more sophisticated implementations, multiple thresholds can be specified, so that a band of intensity values can be set to white while everything else is set to black. For colour or multi-spectral images, it may be possible to set different thresholds for each colour channel, and so select just those pixels within a specified cuboid in RGB space. Another common variant is to set to black all those pixels corresponding to background, but leave foreground pixels at their original colour/intensity (as opposed to forcing them to white), so that that information is not lost.

2.4 Canny Edge Detection:

The purpose of edge detection in general is to significantly reduce the amount of data in an image, while preserving the structural properties to be used for further image processing. The Canny operator[9] was designed to be an optimal edge detector. It takes as input a gray scale image, and produces as output an image showing the positions of tracked intensity discontinuities. The Canny operator works in a multi-stage process. These stages are described in the sections below.

2.4.1 Smoothing:

It is inevitable that all images taken from a camera will contain some amount of noise. To prevent that noise is mistaken for edges, noise must be reduced. Therefore the image is first smoothed by applying a Gaussian filter.

2.4.2 Finding gradients:

The Canny algorithm basically finds edges where the grayscale intensity of the image changes the most. These areas are found by determining gradients of the image. Gradients at each pixel in the smoothed image are determined by applying what is known as the Sobel-operator. First step is to approximate the gradient in the x- and y-direction respectively by applying the kernels shown:

$$K_{GX} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad K_{GY} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

2.4.3 Non- Maximum Suppression:

The purpose of this step is to convert the "blurred" edges in the image of the gradient magnitudes to "sharp" edges. Basically this is done by preserving all local maxima in the gradient image, and deleting everything else. Edges give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output.

2.4.4 Double Thresholding:

The edge-pixels remaining after the non-maximum suppression step are (still) marked with their strength pixel-by-pixel. Many of these will probably be true edges in the image, but some may be caused by noise or colour variations for instance due to rough surfaces. The simplest way to discern between these would be to use a threshold, so that only edges stronger that a certain value would be preserved. The Canny edge detection algorithm uses double thresholding. Edge pixels stronger than the high threshold are marked as strong; edge pixels

weaker than the low threshold are suppressed and edge pixels between the two thresholds are marked as weak.

2.4.5 Edge Tracking by Hysteresis:

The tracking process exhibits hysteresis controlled by two thresholds: *T1* and *T2*, with *T1* > *T2*. Tracking can only begin at a point on a ridge higher than *T1*. Tracking then continues in both directions out from that point until the height of the ridge falls below *T2*. This hysteresis helps to ensure that noisy edges are not broken up into multiple edge fragments.

2.5 Contours:

A contour is nothing but a set of points that represents a curve in the image. Once the edges in an image are detected, we normally would want to arrange the edges into contours. This aids further image processing by providing means to detect specific shapes and extract out the region of interest. Contours are usually represented and stored as sequences of points representing the detected edges.

2.6 Hough Transform:

2.6.1 Standard Hough Transform:

The Hough transform is a technique which can be used to isolate features of a particular shape within an image. As it is necessary, that the desired features be specified in some parametric form, the Standard Hough Transform is most commonly used for the detection of regular curves such as lines, squares, circles, ellipses, *etc.* A generalized Hough transform can be employed in applications where a simple analytic description of features is not possible. Despite its domain restrictions, the Standard Hough Transform retains many applications, as most manufactured parts (and many anatomical parts investigated in medical imagery) contain feature boundaries which can be described by regular curves. The main advantage of the Hough transform technique is that it is tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise.

The Hough transform algorithm uses an array, called an accumulator, to detect the existence of a line y = mx + b. The dimension of the accumulator is equal to the number of unknown parameters of the Hough transform problem. For each pixel and its neighbourhood, the Hough transform algorithm determines if there is enough evidence of an edge at that pixel. If so, it will calculate the parameters of that line, and then look for the accumulator's bin that the parameters fall into, and increase the value of that bin. By finding the bins with the highest values, typically by looking for local maxima in the accumulator space, the most likely lines can be extracted, and their (approximate) geometric definitions are read. The simplest way of finding these peaks is by applying some form of threshold, but different techniques may yield better results in different circumstances - determining which lines are found as well as how many. Since the lines returned do not contain any length information, it is often next necessary to find which parts of the image match up with which lines. The results of this transform get stored in a matrix. Each value represents the number of curves through any point. Higher cell values are rendered brighter. This method reduces the computation time and has the interesting effect of reducing the number of unnecessary tasks, thus enhancing the visibility of the spikes corresponding to real lines in the image.

2.6.2 Probabilistic Hough Transform:

The Probabilistic Hough Transform is defined as the log of the probability density function of the output parameters, given all available input features. It is "probabilistic" because, rather than accumulating every possible point in the accumulator plane, it accumulates only a fraction of them. The idea is that if the peak is going to be high enough , then hitting it only a fraction of the time will be enough to find it ; the result of this conjecture can be a substantial reduction in computation time. A Probabilistic Hough Transform is usually illustrated using the familiar problem of finding straight lines from oriented edges, and it is shown that the conventional Hough method gives a good approximation to the probabilistic method. In situations where there are many unknown parameters, however, conventional methods do not perform well, and here the Probabilistic Hough Transform does provide an effective alternative.

2.7 Colour Identification:

When dealing with images obtained in real-time applications, colour detection and identification can be an important step. Blob detection refers to visual modules that are aimed at detecting points and regions in the image that differ in properties like brightness and colour compared to the surroundings. Blob detection is used when an object of a known colour has to be tracked since it can separate the object to be tracked from the static background. Segmentation is a common method used to partition a digital image into multiple segments (sets of pixels also called superpixels). Each of the pixels in a region are similar with respect to some characteristic property like colour, intensity or texture. Hence, segmentation can be effectively used as a colour identification method.

2.8 K-means Clustering:

The most well-known and commonly used partitioning methods are k-means .The k-means algorithm takes the input parameter k, and partitions a set of n objects into k clusters so that the resulting intra-cluster similarity in high whereas the inter-cluster similarity is low. Cluster similarity is measured in regard to the mean value of the objects in the cluster , which can be viewed as the cluster's "center of gravity". The basic procedure involves producing all the segmented images for two clusters up to Kmax clusters, where Kmax represents an upper limit on the number of clusters. Overall , the k-means method aims to minimize the sum of squared distances between all points and the cluster centre to efficiently favour colour image segmentation.

2.9 Geometry of Lines and Regular Polygons:

Since the Rubik's cube has a well defined geometric shape, the geometrical properties of a cube (lines and squares) have to be known well.

2.9.1 Lines:

A straight line is represented by the following equation where *m* is the slope of the line and *c* is the y-intercept.

$$y = mx + c \qquad \text{.....(1)}$$

The co-ordinates of atleast 2 points in a plane are required to obtain the equation of a line. If (x1, y1) and (x2, y2) are 2 points in a plane, then the line passing through them can be constructed as follows:

1. Find out the slope of the line, m = (y2-y1)/(x2-x1).
2. Find out the y-intercept of the line, c, by substituting m, x1, y1 in equation (1)
3. Substitute m and c in equation (1) to get the equation of the desired line.

To find out if a point lies on a given line, its x and y co-ordinates have to satisfy the equation of the line. The co-ordinates of the point of intersection of two lines are obtained by solving the equations of the 2 lines as follows:

If y = m1.x + c1 and y2 = m2.x +c2 are the equations of the 2 lines, then co-ordinates of their point of intersection, P (px, py) is given by:

py = (m2.c1 – m1.c2)/(m2 – m1)
px = (y – c1)/m1

2.9.2 Regular Polygons:

Polygons are closed figures and are made up of 3 or more vertices. The polygon with 4 vertices is called a quadrilateral. Polygons can be convex or concave. In convex polygon, for each vertex of the polygon, all the other vertices lie on the same side of it. In other words, all internal angles are less than 180 degrees. In a concave polygon, atleast one of the interior angles is always greater than 180 degrees. A regular polygon is one whose all sides (and angles) are equal to each other. A quadrilateral with equal sides is called a rhombus. Properties of a rhombus are necessary to be known if three sides of the Rubik's cube have to be scanned at one time. A

square is a rhombus in which all angles are equal to 90 degrees. The diagonals of a square are equal in length and intersect at 90 degrees.

2.10 OpenCV:

OpenCV[6] is an open source computer vision library which is written in C and C++ and runs under Linux, Windows and Mac OS X. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. OpenCV is written in optimized C and can take advantage of multicore processors. If further optimization is required on Intel architectures, Intel's Integrated Performance Primitives (IPP) libraries can be bought which consist of low level optimized routines in many different algorithmic areas. OpenCV automatically uses the appropriate IPP library at run-time if that library is installed.

One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas in vision, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, and robotics. Because computer vision and machine learning often go hand-in-hand, OpenCV also contains a full, general-purpose Machine Learning Library (MLL). This sub-library is focused on statistical pattern recognition and clustering. The MLL is highly useful for the vision tasks that are at the core of OpenCV's mission, but it is general enough to be used for any machine learning problem.

The open source license for OpenCV has been structured such that one can build a commercial product using all or part of OpenCV. There is- no obligation to open-source one's product or to return improvements to the public domain.

2.11 RGB and HSV colour spaces in digital images:

A colour space is a specific implementation of a colour model *i.e.* colour model is a way of describing and specifying a colour. The term is often used loosely to refer to both a colour space system and the colour space on which it is based. There are many RGB colour spaces. When converted from one colour space to another, any colour that are outside the destination space are "out of Gamut" colours since they can't be represented accurately by the new space. It can checked inorder to see which colour will be either clipped off or re-mapped into the smaller colour space.

Whether colours are clipped or remapped will depend on the rendering intent—which is a term for the mathematical algorithm used to convert from one colour space to another.

RGB is easy to implement but non–linear with visual perception. It is device dependent and specification of colours is semi–intuitive. RGB is very common, being used in virtually every computer system as well as television, video etc. RGB is a colour model that uses the three primary (red, green, blue) additive colours, which can be mixed to make all other colours. RGB builds its model on different colours of light added together, where the mixture of all three colours produces white light. HSV – Hue Saturation Value is based on the artist concepts of Tint, Shade, and Tone respectively. The HSV colour space, like RGB, is a device-dependent colour space, meaning the actual colour you see on your monitor depends on what kind of monitor you are using, and what its settings are. The terms Hue, Saturation and Value are defined as :

- Hue: This is the colour itself, which results from the combination of primary colours. All shades (except for the gray levels) .The term "colour" is often used instead of "Hue". The RGB colours are"primary colours".
- Saturation in HSV: It is that quality of colour by which we distinguish a strong colour from a weak one; the degree of departure of a colour sensation from that of a white or gray; the intensity of a distinctive hue; colour intensity.
- Value: It is that quality by which we distinguish a light colour from a dark one.This value describes the luminosity , the luminous intensity. It is the amount of light emitted by a colour.

To transform an RGB image into the HSV (hue, saturation, and value) colour space. The input RGB values must be byte data in the range 0 to 255. An input file with at least 3 bands or a colour display must be open to execute this function. The stretch that is applied in the colour display is applied to the input data. The hues produced are in the range of 0 to 360 degrees (where red is 0 degrees, green is 120 degrees, and blue is 240 degrees) and saturation and value in the range 0 to 1 .

2.12 Rubik's cube and Solving Methods:

The Rubik's cube[8] is a 3-D mechanical puzzle invented by Hungarian Sculptor, Erno Rubik in 1974. In a classic Rubik's Cube, each of the six faces is covered by nine stickers, among six solid colours (white, red, blue, orange, green, and yellow). A pivot mechanism enables each face to turn independently, thus mixing up the colours. For the puzzle to be solved, each face must be a solid colour.

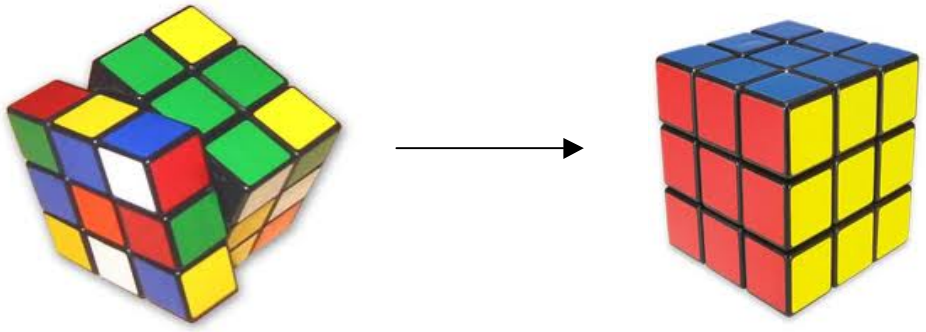

Figure 2.2: Rubik's cube

In Rubik's cubists' parlance, a memorised sequence of moves that has a desired effect on the cube is called an algorithm. This terminology is derived from the mathematical use of algorithm, meaning a list of well-defined instructions for performing a task from a given initial state, through well-defined successive states, to a desired end-state. Each method of solving the Rubik's Cube employs its own set of algorithms, together with descriptions of what the effect of the algorithm is, and when it can be used to bring the cube closer to being solved.

Most algorithms are designed to transform only a small part of the cube without scrambling other parts that have already been solved, so that they can be applied repeatedly to different parts of the cube until the whole is solved.

Some algorithms have a certain desired effect on the cube but may also have the side-effect of changing other parts of the cube. Such algorithms are often simpler than the ones without side-effects, and are employed early on in the solution when most of the puzzle has not yet been solved and the side-effects are not important. Towards the end of the solution, the more

specific algorithms are used instead, to prevent scrambling parts of the puzzle that have already been solved.

Although there are a significant number of possible permutations for the Rubik's Cube, a number of solutions have been developed which allow for the cube to be solved in well under 100 moves. The most popular solution involves solving the Cube layer by layer, in which one layer (designated the top) is solved first, followed by the middle layer, and then the final and bottom layer.

2.13 Open GL:

OpenGL is a software interface to graphics hardware. The interface consists of over 700 commands (about 650 in core OpenGL and another 50 in OpenGL utility library) that can be used to specify the objects and operations needed to produce interactive 3 dimensional applications.

OpenGL is designed as streamlined hardware independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead one must work through the windowing system that controls the particular hardware being used. Similarly OpenGL does not provide high level commands for describing models of 3 dimensional objects. Such commands permits one to specify relatively complicated shapes such as molecules, parts of body, automobiles, etc. with OpenGL building ones desired model from a small set of geometric primitives – points, lines and polygons is viable.

The following list briefly describes the major graphics operations that OpenGL performs to render an image on the screen.

1. Construct shapes form the geometric primitives, thereby creating mathematical description of objects, OpenGL considers points, lines, polygons, images and bitmaps to be primitives.
2. Arrange the objects in a 3 dimensional space and select the vantage point for viewing the composed scene.

3. Calculate the colours of all objects. Colours might be explicitly assigned by the application, determined from specific lighting conditions, obtained by passing a texture onto the objects, or some combination of these 3 actions.
4. Convert the mathematical description of objects and their associated colour information to pixels on the screen. This process is called rasterization.
   During this stage OpenGL might perform other operations such as eliminating parts of objects that are hidden by other objects. In addition, after the scene is rasterized, but before it is drawn on the screen, you can perform some operations on the pixel data if you want.

   OpenGL is also designed to work if the computer that displays the graphics being created is not the one that runs the program. This could be the case of work in a networked computer environment where many computers are connected to one another by a digital network . In this case, the computer on which your program runs and issues OpenGL drawing commands is called the client and the computer that receives those commands and performs the drawing is called the server. The format for transmitting OpenGL commands (called the protocol) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers.

2.13.1 Few important terms:

1. **Rendering**: It is the process by which a computer created images from models. These models are constructed from geometric primitives that are specified by the vertices.
2. **Pixel**: it is the smallest visible element that the display hardware can put on the screen.
3. **Bitplanes**: it is an area of the memory that contains one bit of information for every pixel on the screen.
4. **Framebuffer**: it holds all the information that the graphics display needs to control the colour and the intensity of all the pixels on the screen.

2.13.2 OpenGL as a state machine:

OpenGL is a state machine. It must be put into various states that then remain in effect until changes are being made. For example, the current colour is state variable. It can be set to red, black, white, etc and thereafter every object is drawn with that colour until the current colour is set to something else. Other variables control things such as current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, positions and characters of lights, material properties of the objects being drawn, etc. Each state variable has a default value and at any point the system can be queried for the current value.

OpenGL rendering pipeline:

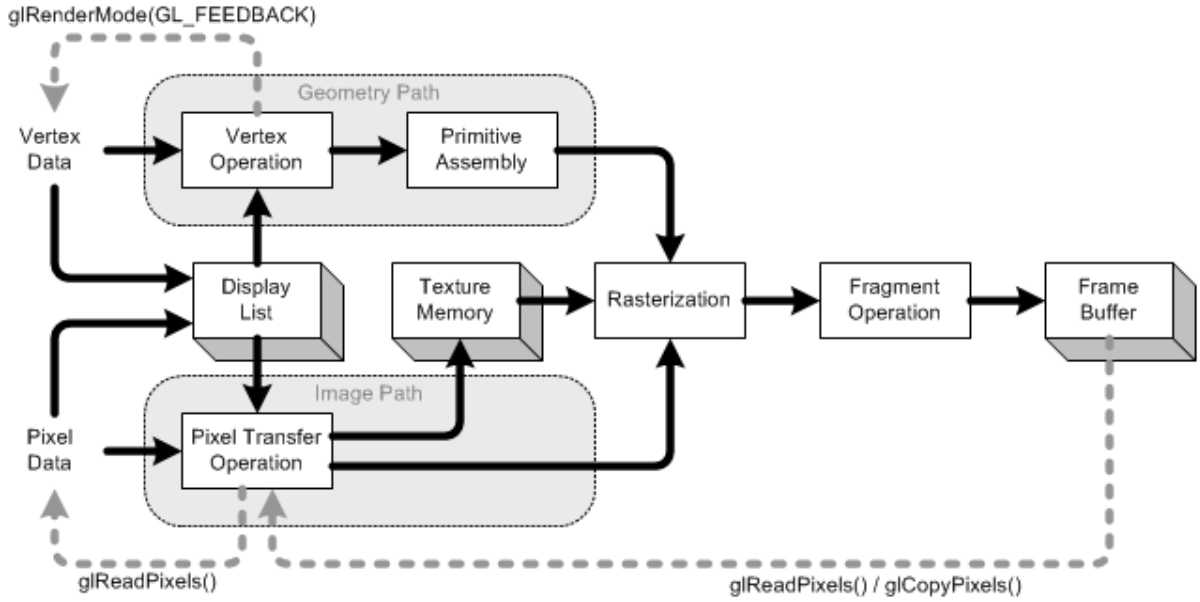

Figure2.3: OpenGL rendering pipeline

- **Display list:** all data whether it describes geometry or pixels can be saved in a display list for current or later use. When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.
- **Evaluators:** all geometric primitives are eventually described by vectors. Parametric curves and surfaces may be described by control points and polynomial functions called basis functions. Evaluators provide a method for deriving vertices used to represent the surface from the control points. The method is a polynomial mapping which can produce surface normal, texture coordinates, colours and spatial values from the control points.

- **Per-vertex operations:** for vertex data, next is the pre-vertex operations stage which converts vertices into primitives. Some types of vertex data are transformed by 4x4 floating point matrices. Spatial coordinated are projected from a position in the 3D world to a position on your screen. If advanced featured like texturing and lighting are enabled, then this stage can get busier.
- **Primitive assembly:** clipping is the elimination of the portion of geometry that falls outside the half space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping add additional vertices depending on how the line or polygon is clipped. In some cases, this is followed by perceptive division which makes distant objects appear smaller than closer objects. The viewport and depth operations are applied. The results of this stage are complete geometric primitives which are the transformed and clipped vertices with related colour, depth and sometimes texture-coordinate values and guidelines for the rasterization step.
- **Pixel operations:** pixels from an array in system are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased and processed by a pixel map. The results are clamped and the either written into texture memory or sent to the rasterization step.
- **Texture assembly:** OpenGL applications can apply texture images onto geometric objects to make them look more realistic. Some OpenGL implementations might have special resources to accelerate texture performance.
- **Rasterization:** it is the conversion of geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the frame buffer. Line and polygon stipples, line width, point size, shading model and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Colour and depth values are assigned for each fragment.
- **Fragment operations:** before values are actually stored in the frame buffer, a series of operation are performed that may alter or even throw out fragments. All those operations can be enabled or disabled. These operations include texturing, fog calculations, scissor test, alpha test, stencil test, depth buffer test, blending, dithering, logical operations and masking by a bitmask.

2.13.3 OpenGL related libraries

1. **OpenGL utility library (GLU):** It contains several routines that use low level OpenGL commands to perform tasks such as setting up matrices for specific viewing orientations and projections, performing polygon tessellation and rendering surfaces. This library is provided as part of every OpenGL implementation.
2. **OpenGL utilty toolkit (GLUT):** GLUT is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of different window APIs.
   - **Window management:**
     - **glutInit():** initializes GLUT to process any command line arguments
     - **glutDisplayMode():** specifies whether to use an RGBA or colour-index colour model.
     - **glutInitWindowPosition():** specifies screen location for upper left corner of the window.
     - **glutInitWindowSize():** specifies size in pixels of the window.
     - **glutCreateWindow():** creates a window with and OpenGL context. Returns a unique identifier for the new window.
   - **display callback:**
     - **glutDisplayFunc():** most important event callback function. Whenever GLUT determines that the contents of the window have to be changed, the callback function registered by this method is executed.
     - **glutPostRedisplay():** if the program changes the contents of the window, them there may be a necessity to call this method. It gives a nudge to the glutMainLoop() method to call the registered display callback at its next opportunity.
   - **Running the program:**
     - **glutMainLoop():** this should be called at the very end. All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins and the registered display callback is triggered. Once this loop is entered, it is never exited.
   - **Handling input events:**

- **glutReshapeFunc():** It indicates what action should be taken when the window is resized.
- **glutKeyboardFunc()** and **glutMouseFunc():** these methods allow to link a keyboard key or a mouse button with a routine that's invoked when the key or mouse button is pressed.
- **glutMotionFunc():** it registers a routine to call back when the mouse is moved while a mouse button is also pressed.

- **Managing a background process:**
  - **glutIdleFunc():** One can specify a function to be executed if no other events are pending with this method. The routine takes a pointer to a function as its only argument. Pass NULL to disable the execution of this function.
- **Drawing 3 dimensional objects:** GLUT includes several routines for drawing three dimensional objects either as wire frames or as solid shaded objects with surface normals defined. For example, the following routines are used to draw a cube:
  - **glutWireCube():**
  - **glutSolidCube():**

  Similar methods are used to draw sphere, cone, icosahedrons, teapot, tetrahedron, torus, dodecahedron and octahedron.

2.13.4 Animation:

One of the exciting things you can do on a graphics computer is draw pictures that move. Hence animation is clearly an important part of computer graphics.

In a movie theatre, motion is achieved by taking a sequence of pictures and projecting them at 24 frames per second on the screen. Each frame is moved in position behind the lens, the shutter is opened and the frame is displayed. The shutter is momentarily closed while the frame is advanced to the next frame, then the frame is displayed and so on. Although 24 different frames are watched each second, ones brain blends them all into a smooth animation.

The key reason why motion picture projection works is that each frame is complete when it is displayed. But in computer animation, the time it takes for your system to clear the screen and to

draw a typical frame. The results are increasingly poor depending upon how close to 1/24th second the system takes to clear and draw.

Most OpenGL implementations provide double-buffering-hardware or software that supplies 2 complete colour buffers. One is displayed while the other is being drawn. When the drawing of a frame is complete, the 2 buffers are swapped, so the one that was being viewed is now used for drawing and vice versa. With double-buffering, each frame is shown only when the drawing is complete: the viewer never sees a partially drawn frame.

MOTION = REDRAW + SWAP

Usually it is easy to redraw the entire buffer from scratch for each frame than to figure out which parts require redrawing. This is especially true with applications like 3D flight simulator where a tiny change in the flight's orientation changes the position if everything outside the window. In most animations, the objects are simply redrawn with different transformations – the viewpoint of a viewer moves or an object is rotated slightly. If significant re-computation is required for non-drawing operations, then the attainable frame rate often slows down.

2.13.5 The Camera Analogy

The transformation process used to produce the desired scene for viewing is analogous to taking a photograph with a camera. The steps with a camera might be the following:

1. Set up the tripod and point the camera in the direction of the scene. (viewing transformation)
2. Arrange the scene to be photographed into the desired composition (modeling transformation)
3. Choose a camera lens or adjust the zoom (projection transformation)
4. Determine how large the final photograph is to be like (viewport transformation)

These steps correspond to the order in which you specify the desired transformations on the program and not necessarily the order in which the relevant mathematical operations are performed on the object's vertices.

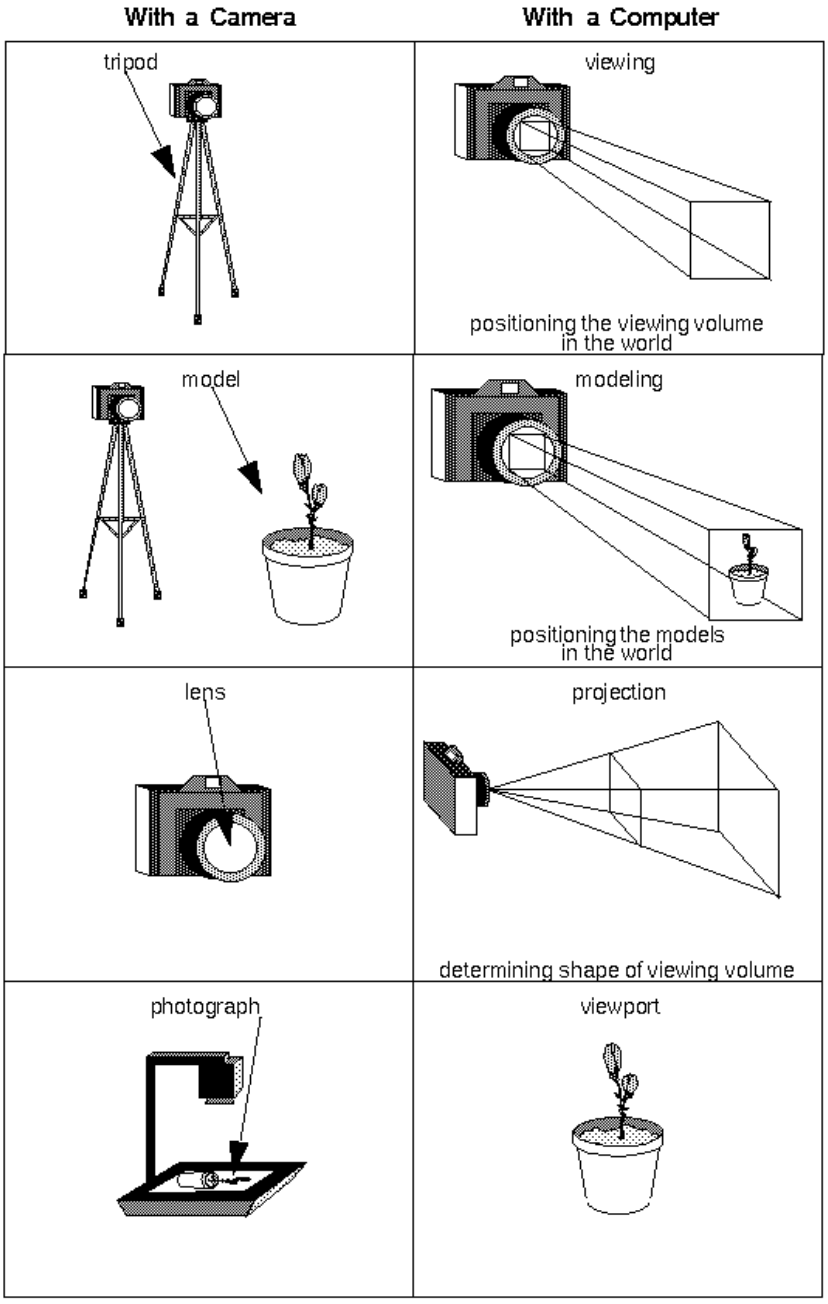

Figure2.4: The camera analogy

The viewing transformation: This is analogous to positioning and aiming a camera. The position of the camera (or eye position), the point that the camera is aimed at and the orientation of the camera have to be specified. This is done by using the gluLookAt() function.

The modeling transformation: One can use modeling transformation to position and orient the model. For examples rotating, translating or scaleing the model – or perform some other

combination of these operations. Note that instead of moving the camera to view the model (using a viewing transformation), the model can be moved away from the camera (using a modeling transformation).

The projection transformation: Specifying the projection transformation is like choosing a lens for the camera. This transformation could be thought of as determining what field of view or viewing volume is and therefore what objects are inside it and to some extent how they look. This is equivalent to choosing among wide angle, normal or telephoto lenses. In addition to the field-of-view considerations, the projection transformation determines how objects are projected onto the screen. Two basic transformations are provided by OpenGL along with several corresponding commands for describing the relevant parameters in different ways. One type is the **perspective** projection, which matches how one can see things in daily life. It makes objects further away , look smaller. The other type of projection is **orthographic,** which maps objects directly onto the screen without affecting their relative sizes.

The viewport transformation: Together the projection transformation and the viewport transformation determine how a scene is mapped onto the computer screen. The projection transformation specifies the mechanics of how the mapping should occur, and the viewport indicates the shape of the available screen area into which the scene is mapped. Since the viewport specifies the region the image occupies on the computer screen, viewport transformation could be thought of as defining the size and location of the final processed photograph.

Drawing the scene: Once all the necessary transformations have been specified, the scene can be drawn. As the scene is drawn, OpenGL transforms each vertex of every object in the scene by the modeling and viewing transformations. Every vertex is then transformed as specified by the projection transformation and clipped if it lies outside the viewing volume described by the projection transformation. Finally the remaining transformed vertices are mapped to the viewport.

CHAPTER 3

PROBLEM STATEMENT

The project deals with developing an application that allows the user to let the system scan an unsolved (also called scrambled) Rubik's cube and determine the next steps to be taken to solve it. This can be divided into 2 sub-problems:

- Identifying the current configuration of the Rubik's cube by processing the frames obtained from a webcam which contain the images of the six faces of the cube.
- Applying the Rubik's cube solving algorithms on the identified configuration to generate and display steps to solve the cube.

Rubik's cube is a deceptively simple-looking puzzle. It represents a difficult problem in computation, with over a quintillion different permutations, finding the move combinations to solve a given cube can be very intensive. Rubik's cubes are typically a cube with a 3x3 grid on each face. Variations on the Rubik's Cube, both physical and virtual, have varied on the original design to change the size of each face's grid as well as to add dimensions to the underlying shape. In its solved state, each of the sides is made up of tiles of the same color, with a different color for each side. Each of the tiles is part of a small cube, called a "cubie." Each face of the cube (made up of nine cubies) can be rotated. The mechanical genius of the puzzle is that the same cubie can be rotated from multiple sides. The basic goal is to take a cube whose sides have been randomly rotated and figure out how to get it back to the solved state. A lot of work has already been done towards creating algorithms that find the optimal solution. But the conveyance medium to the user for efficiently learning of solving the cube has not been implemented with core concerns of the cubist, inspite of the vast development with technology.

Computer Vision techniques are used to make the interface between the user and computer simpler from the user's point of view, and thus, make it more appealing to them than the existing applications that require a lot more user interaction just to identify the cube configuration.

CHAPTER 4

SYSTEM REQUIREMENT SPECIFICATION

Software requirement Specification is a fundamental document, which forms the foundation of the software development process. It not only lists the requirements of a system but also has a description of its major feature. An SRS is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies at a particular point in time (usually) prior to any actual design or development work. It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

4.1 User Requirement Specification

This lists out in clear terms, exactly what functionalities the user requires from the system. It also lists out the non- functional requirements i.e. some of the implied features that the system should possess in addition to the user- stated functionalities.

4.1.1 Functional Requirements

- A convenient method user to input the Rubik's cube configuration to the computer requiring minimal interaction with the system.
- Rubik's cube solving steps should be presented in an easily understandable manner.

4.1.2 Non Functional Requirements

- **Ease of use**: The application's ease of use vastly affects user satisfaction. Easier the usage, more the satisfaction level. In our application we try as much as possible to minimize the number of keystrokes required for the user to input the Rubik's cube configuration.
- **Accuracy**: The application should accurately detect the Rubik's cube, identify the sticker colours, understand the cube's configuration and show the correct steps to solve it.
- **Speed**: The application should be able to deliver results (show the solution steps) with minimum time delay.

4.2 Hardware Requirement Specification:

- Rubik's cube
- Desktop/Laptop running Windows XP or Windows 7 operating system
- Camera (webcam is good enough) – this is required to serve as the sensor that provides input to the application.
- RAM – 1 GB
- Processor – Intel Core II Duo (1.73 GHz)
- Keyboard – standard 102 keys

4.3 Software Requirement Specification:

4.3.1 Visual Studio 2008 Express Edition:

Microsoft Visual Studio Express is a set of freeware integrated development environments (IDE) developed by Microsoft that are lightweight versions of the Microsoft Visual Studio product line. The idea of Express editions is to provide streamlined, easy-to-use and easy-to-learn IDEs for users other than professional software developers, such as hobbyists and students. The Visual C++ Express Edition can be used to compile .NET as well as Win32 applications.

4.3.2 OpenCV 2.3.0:

OpenCV is a free open source library of programming functions mainly aimed at real time computer vision. It has a BSD license (free for commercial or research use). OpenCV was originally written in C but now has a full C++ interface and all new development is in C++. There is also a full Python interface to the library.

4.3.3 OpenGL :

OpenGL a software interface to graphics hardware. OpenGL is designed as streamlined hardware independent interface to be implemented on many different hardware platforms. The purpose of OpenGL is to communicate with the graphics card about 3D scene . Hence , it serves as a translator for graphics card.

CHAPTER 5

DATA FLOW DIAGRAM

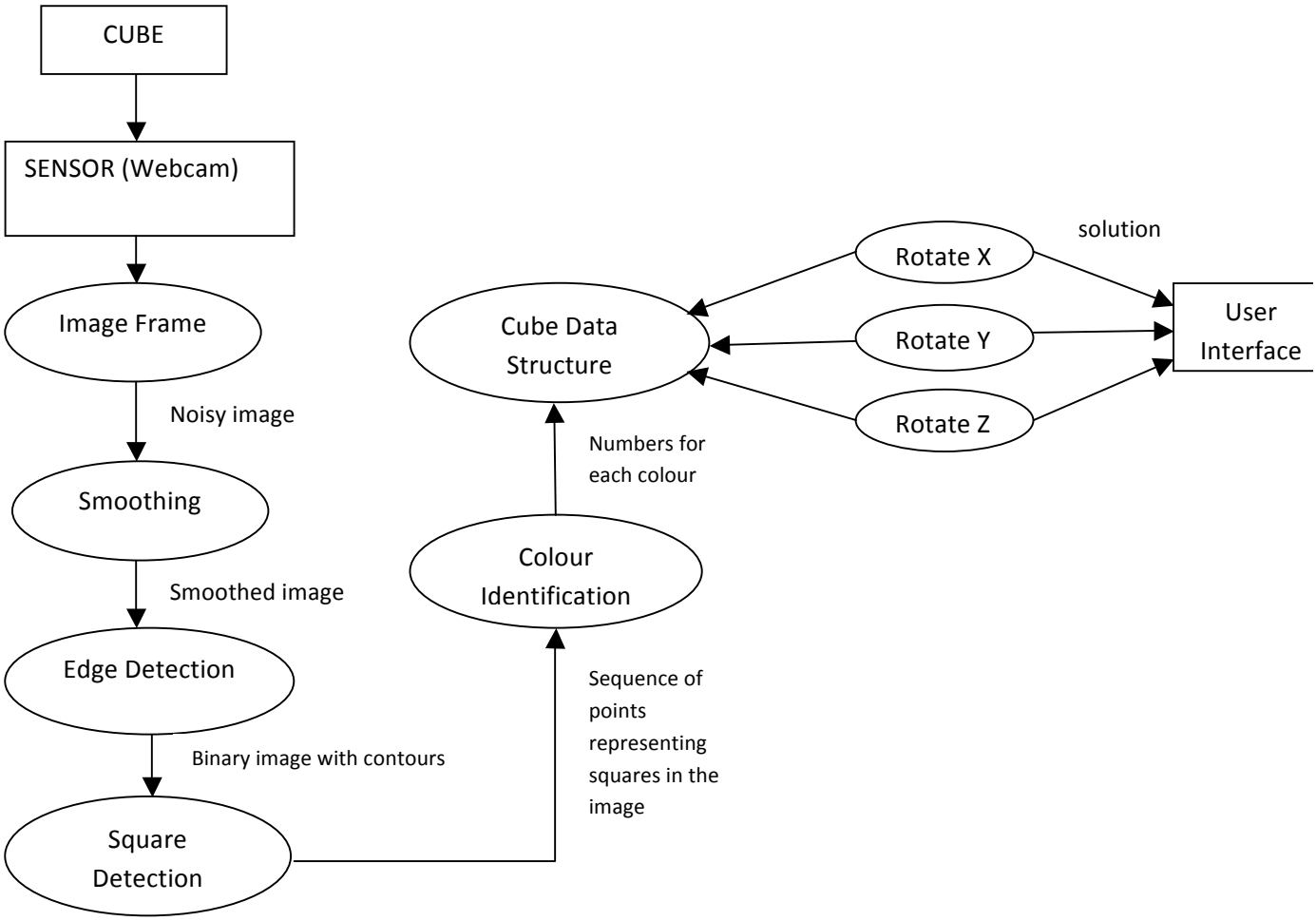

Figure 5.1: Data Flow Diagram

The system has been divided into two main functional parts – the first part deals with identification of the Rubik's cube and reading its configuration; the second part involves application of Rubik's cube solving algorithms to display the solution steps.

5.1 Cube Configuration Detection

The first part of the application involves identification of the cube configuration. The following processes come into play:

- Input is received from the sensor (webcam) in the form of image frames. Six frames contain images showing the 6 faces of the cube.
- **Smoothing**: this is applied to each frame to reduce the noise.
- **Edge detection:** canny edge detection is used to detect all the edges and strong contours in the image
- **Square Detection:** approximation of contours to polygons and filtering out only the ones those polygons that can be squares in the image
- **Colour Identification:** the colour of each square is identified , and grouped six clusters based on hue of their colours.

5.2 Rubik's Cube Solving

The second part of this application deals with solving of the Rubik's cube. The following processes come into play:

- **Rotate X:** Rotation of a layer of the cube in either left or right direction
- **Rotate Y:** Rotation of a layer of the cube in either upward or downward direction
- **Rotate Z:** Rotation of a layer of the cube in either clockwise or anti-clockwise direction

These processes perform various transforms on the cube data structure and output the rotations as solution steps to the user interface

CHAPTER 6

SYSTEM DESIGN AND IMPLEMENTATION

The scrambled cube is held by the user and the state of the cube is video streamed through the sensor such as webcam. The image frame being captured undergoes smoothing, which is later subjected to edge detection and removal of outliers to identify and obtain only the necessary line segments in the image. Finally, Histogram segmentation and colour identification methods are applied to obtain an efficient image which enables application of the algorithm. After the state being recognized, it is given as input to the cubes data structure and later decides with the steps adopted to solve the cube completely. The moves happen through the rotate functions depending on which direction the layers of the cube have to be rotated to obtain a completely solved cube.

6.1 System Design

6.1.1 High Level Design

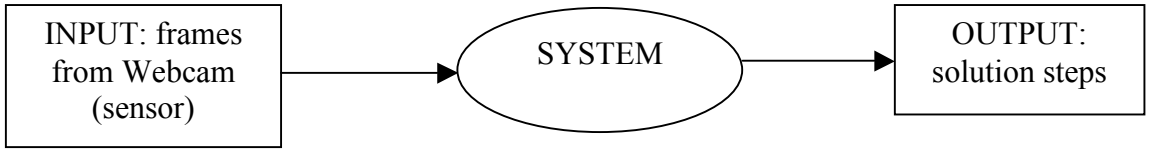

Figure 6.1: Level 1 DFD

High level software design, also called software architecture is the first step to analyze and consider all requirements for a software and attempt to define a structure which is able to fulfill them. As shown in the figure 6.1, at the highest level of abstraction, the system takes input from the webcam which is essentially frames from the video stream. A total of 6 frames are obtained, one for each face of the Rubik's cube. The output of the system of the solution steps to solve the Rubik's cube.

The system's total functionality is divided into 2 main segments:

- **Identification of cube configuration**- this segment deals purely with Computer Vision techniques required to derive the required knowledge about the configuration of the

Rubik's cube. Input to this will be the image frame from the webcam and output is an array of numbers that represent each colour on the cube face. This is used to populate the cube data structure which represents the cube configuration in a simplified manner so that the second segment can conveniently apply its functionality on it.

- **Solving the cube** – this segment purely deals with solving of the Rubik's cube. The technique used to solve the cube is called the layered approach. In this approach, we attempt to solve the cube one layer at a time. Rotation functions applied to the cube data structure (input) yield the subsequent steps that the user must follow to solve the cube. These steps are displayed as output.

6.1.2 Architectural Strategies

The Rubik's cube problem solver here is designed considering the cubists' requirements and concerns over learning to solve the cube. The state of the cube is video streamed through a sensor such as webcam which is the input given to the system to decide on the way the algorithm is to be proceeded to obtain the solved cube with displaying (on appropriate user interface) the steps for the user to solve it themselves.

An iterative model has been followed to develop this application. The two segments described above have been developed separately, each involving a set of modules. Each of these modules were refined to obtain better results. Careful designing of the cube data structure has ensured that the two segments integrate seamlessly and the task is accomplished. Sequential programming such that modules work with assistance of various helper functions has been followed.

This approach has also made sure that the application can be extended to non – standard Rubik's cubes i.e. cubes which have stickers of colours other than the standard colours. Also, it can be extended to higher level cubes as well.

6.1.3 Block diagram

The functionality of the application is depicted in the following block diagram:

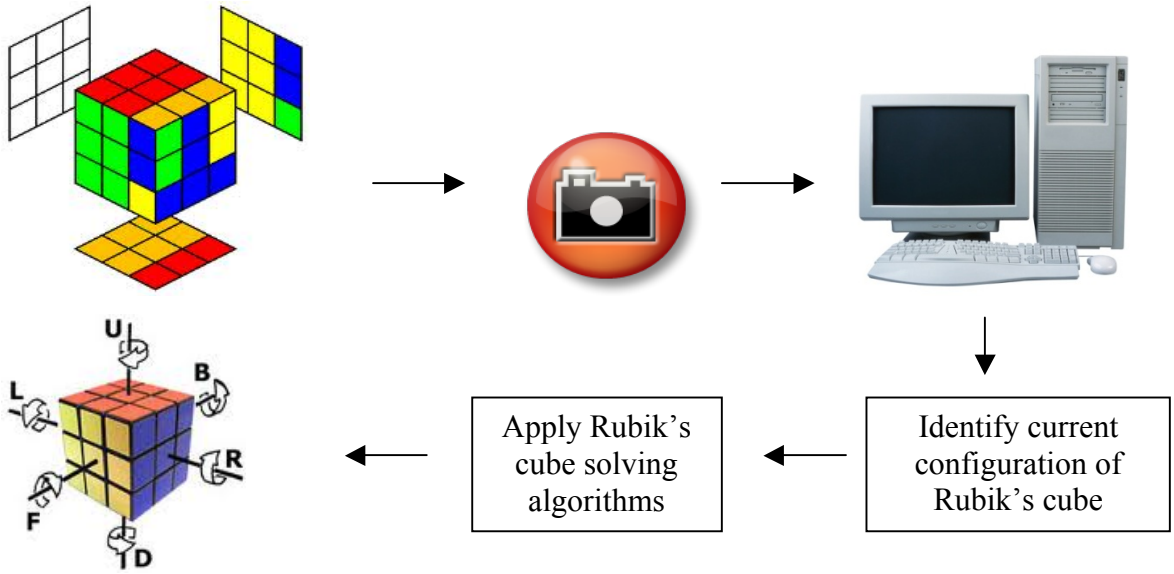

Figure 6.2: Block diagram of Rubik's Cube Problem Solver

6.2 Implementation Details

The 3x3x3 Rubik's cube solver consists of two major parts to obtain the solved state in the system's phase being indicted in the *Figure 6.3*. One is the object detection phase and the other is the application of the algorithm based on the detected object.

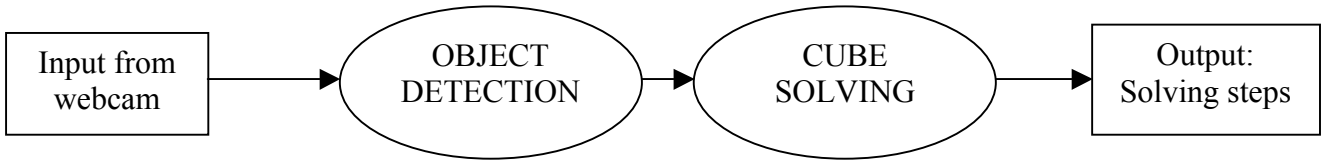

Figure 6.3: Parts of the system

6.2.1 Phase 1 – Object Detection

In the first phase of object detection the state of the cube is been captured through the webcam. The image obtained is subjected to smoothing, followed by edge detection (Canny edge detection) to obtain all the contours necessary in the frame to get the colours of the stickers on each face. Since all the edges are present, there is a necessity to apply a lot of filtering conditions to get only the squares in the cube. Finally, through colour detection, the colour of each sticker is extracted for further computation.

Modules used in this part of the application are as follows:

- **double angle(CvPoint* pt1, CvPoint* pt2, CvPoint* pt0)**
  find out the ange between vectors from point pt0 to pt1 and from pt0 to pt2

- **void intersection_point(CvPoint *one, CvPoint *two, CvPoint *three, CvPoint *four, CvPoint *intersect)**
  function to fing intersection point of two lines. Points one and two are of the first line, points three and four are of the second line

- **float distance (CvPoint *one, CvPoint *two)**
  function to find the distance between two points

- **void find_centre_points(CvSeq * squares, CvSeq *centres)**
  funtion to find and store alll 9 points of intersection in a sequence

- **void order_points(CvSeq *centers)**
  function to get the points in order so that they can be populated into the cube data structure

- **void drawSquares(IplImage* img, CvSeq* squares)**
  function to return sequence of squares detected on the image. The sequence is stored in the specified memory storage

- **CvSeq* findSquares4(IplImage* img, CvMemStorage* storage)**
  This function draws all the squares in the image

  - **void average_pixel_colour(IplImage *img, CvPoint *pixel, int n)**
    function to average colour of pixels , and the average is calculated over (2n+1)*(2n+1) grid surrounding pixel

  - **void intersection_point(CvPoint *one, CvPoint *two, CvPoint *three, CvPoint *four, CvPoint *intersect)**

function to find intersection point of 2 lines . Points one and two are of the first line, points three and four are of the second line intersect holds the point of intersection

- **CvSeq* squares = cvCreateSeq(0, sizeof(CvSeq), sizeof(CvPoint), storage)**
  This is to create empty sequence that will contain points - 4 points per square (the square's vertices)

- **cvSetImageROI(timg, cvRect(0, 0, width, height))**
  select the maximum ROI(Region Of Interest) in the image with the width and height divisible by 2

- **cvSetImageCOI(image, 0)**
  This is to extract the c-th color plane by selecting the COI (Channel Of Interest)

6.2.2 Phase 2 – Rubik's Cube Solving

In this phase the final output ie the steps to solve the Rubik's cube are displayed. Each of the identified colours is represented as numbers from 1-6. The colour of the centre cubie determines the colour of the face. Each face of the cube is represented as a structure consisting of a 3x3 array and pointers to its top, bottom, left and right faces. The structure is shown below:

```
struct face
{
        int arr[3][3];
        struct cube *u;
        struct cube *l;
        struct cube *r;
        struct cube *d;
};
```

The modules used are as follows:

- **void rotate_x(int dir,int index)**

function to rotate the faces in x-y plane i.e about the y-axis by 90 degrees. dir=0 denotes rotation towards the right and dir=1 denotes rotation towards the left. index takes values 0,1 or 2 which determines whether the top, middle or the bottom face has to be rotated respectively.

- **void rotate_y(int dir,int index)**
  function to rotate the faces in y-z plane i.e. about the x axis. dir=0 denotes upward rotation and dir=1 denotes downward rotation. index takes values 0,1 or 2 which determines whether the left, middle or the right face has to be rotated respectively.

- **void rotate_z(int dir,int index)**
  function to rotate the faces in x-y plane i.e about the z axis. dir=0 denotes clockwise rotation and dir=1 denotes anti-clockwise rotation. index takes values 0,1 or 2 which determines whether the back, middle or the front face has to be rotated respectively.

- **int* change_left(int *pS)**
  changes the values on the faces caused due to rotation

- **int* change_right(int *pS)**
  changes the values on the faces caused due to rotation
- **void showXLayer0(void);**
  the graphics function to display the 1st layer (right most layer) of the cube that lies in the y-z plane and that can be rotated about the x axis
- **void showXLayer1(void);**
  the graphics funtion to display the 2nd layer (middle layer) of the cube that lies in the y-z plane and that can be rotated about the x axis.
- **void showXLayer2(void);**
  the graphics funtion to display the 3rd layer (right most layer) of the cube that lies in the y-z plane and that can be rotated about the x axis.
- **void showYLayer0(void);**
  the graphics funtion to display the 1st layer (top most layer) of the cube that lies in the x-z plane and that can be rotated about the y axis.

- **void showYLayer1(void);**
  the graphics funtion to display the 2nd layer (middle layer) of the cube that lies in the x-z plane and that can be rotated about the y axis.
- **void showYLayer2(void);**
  the graphics funtion to display the 3rd layer (bottom most layer) of the cube that lies in the x-z plane and that can be rotated about the y axis.
- **void showZLayer0(void);**
  the graphics function to display the 1st layer (front layer) of the cube that lies in the x-y plane and that can be rotated about the z axis.
- **void showZLayer1(void);**
  the graphics function to display the 2nd layer (middle layer) of the cube that lies in the x-y plane and that can be rotated about the z axis.
- **void showZLayer2(void);**
  the graphics function to display the 3rd layer (back layer) of the cube that lies in the x-y plane and that can be rotated about the z axis.
- **void displayAll(void);**
  the graphics function to display the complete cube.
- **void getCubieColor(const struct cube *face, const int x, const int y, float *r, float *g, float *b);**
  function to retrieve the colour of the cubie on "face" at (x, y) position from the cube structure and set the RGB values for displaying on the screen.
- **void myDisplay(void);**
  the OpenGL display function that includes all routines to redraw as scene on the screen.
- **void myInit(void);**
  function that includes all the initializations required before graphics rendering can be done like setting the clearing colour.
- **void myMouseFunc(int x, int y);**
  the mouse event handler that handles mouse click input events. It is registered with glutMouseFunc().
- **void myKeyboardFunc(unsigned char key, int x, int y);**
  the keyboard event handler that handles key stoke input events. It is registered with glutKeyboardFunc().

- **void myReshape(void);**
  event handler to handle window resize input event. It is registered with the glutReshapeFunc() and resets the size of the viewport.
- **void myIdleFunc(void);**
  this function is called during the idle time after the display function finishes executing and is used to change the value of angle of rotation when the rotation of the cube has to be shown in animation. It then calls the glutPostRedisplay() function which gives glutMainLoop() a nudge to call the myDisplay() function.

A layered approach is use in solving the cube.:

Step 1: Formation of a '+' pattern on the top face such that the edge cubies match the adjacent faces' centre cubie

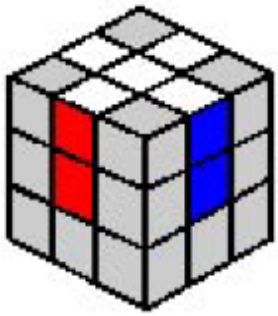

Figure 6.4: Configuration of the cube after the first step

Step 2: Completion of the top face and the first layer by solving for the corner cubies

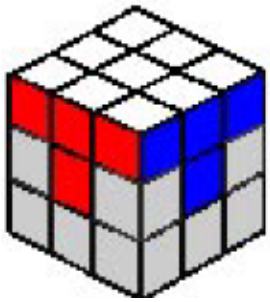

Figure 6.5: Configuration of the cube after the second step

Step 3: Completion of the second layer by solving for the egde cubies in the middle layer

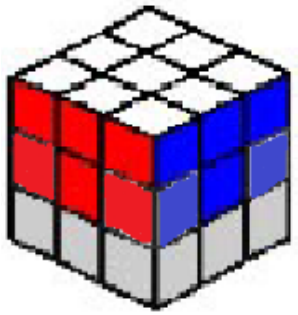

Figure 6.6: Configuration of the cube after the third step

Step 4: Solving the last layer using specific algorithms

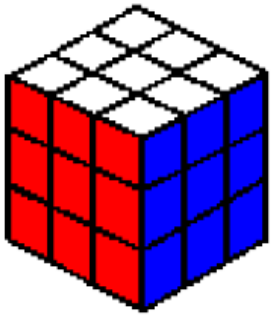

Figure 6.7: The solved cube

6.2.3 Testing

- **Program metrics :**

  Lines of code : The total lines of code for the Rubik's cube solving application is around 5500 lines.

- **Functional metrics :**

  Considering the number of moves to solve the cube completely, the minimum number of moves to obatin a completely solved cuve is 0 while the maximum moves are 278.

  In the layered approarch being adopted, various conditions are to be considered and checked inorder to postion or move a cube. Totally, the program consists of around 278 conditions being considered and checked.

CHAPTER 7

SIMULATION, SYNTHESIS AND IMPLEMENTATION RESULTS

The application is tested under different lighting conditions to check if the cubies and the sticker colours are correctly identified. It is also tested to check if the correct moves are displayed for different configurations of the cube. The results at various stages are listed below.

7.1 Simulation results

Phase 1: The 6 different faces of the cube are captured, edges are indentified, squares are detected and the colours on all the cubies are identified.

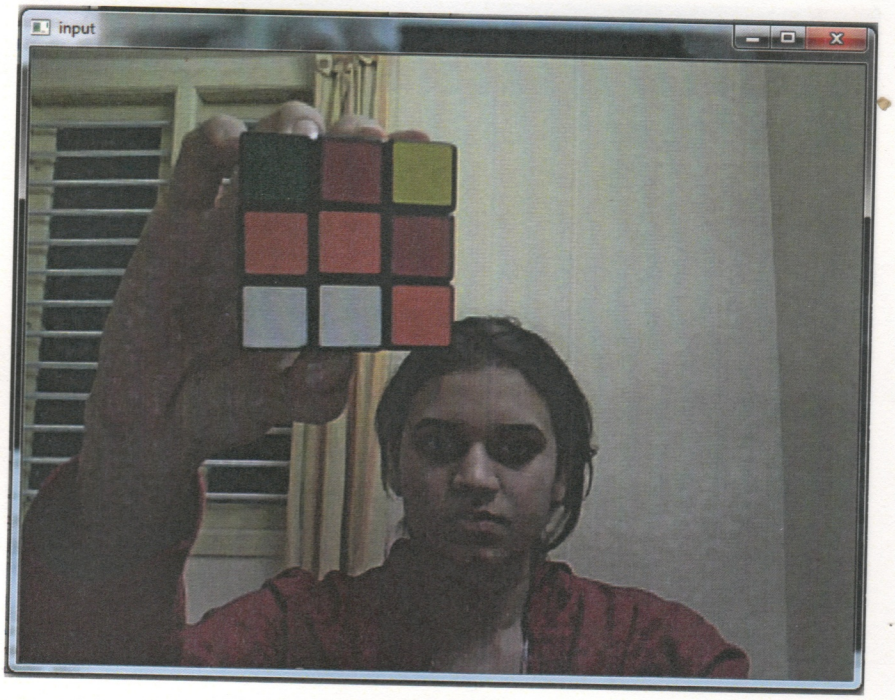

Figure 7.1: input image showing one of the faces of the cube

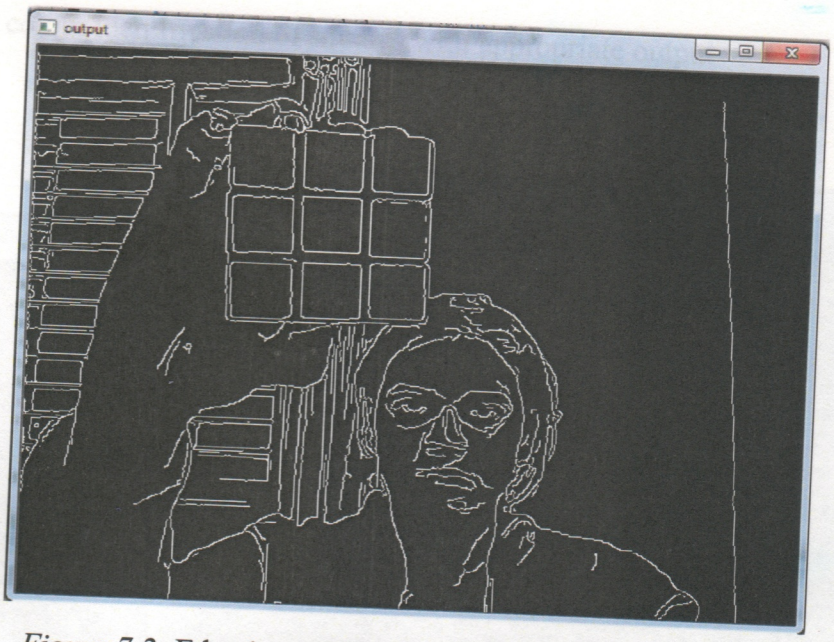

Figure 7.2: Edge image for applying Canny edge detection

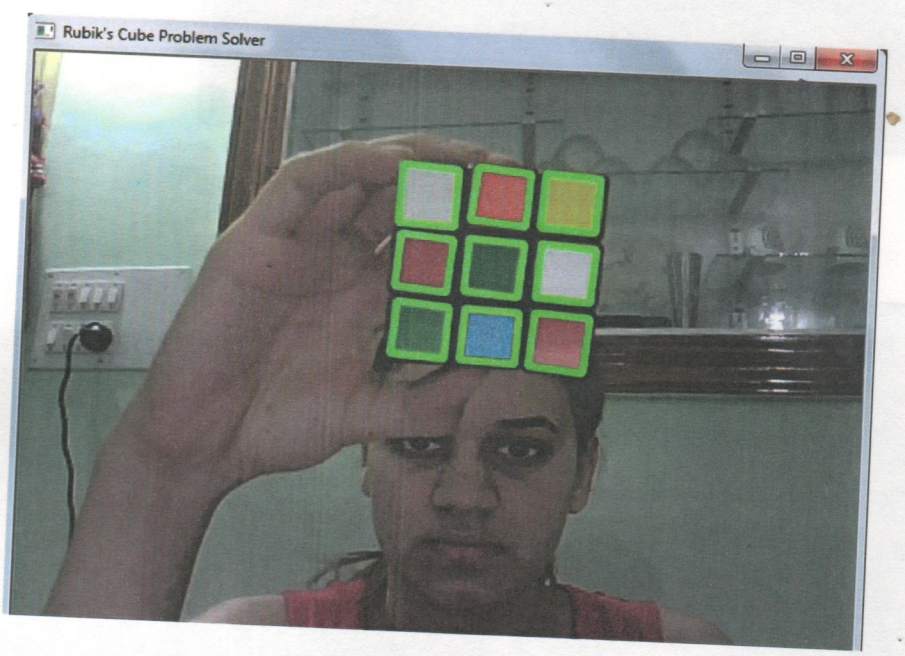

Figure 7.3: Image showing the cubies identified

Phase 2: The colour identification of squares with appropriate output

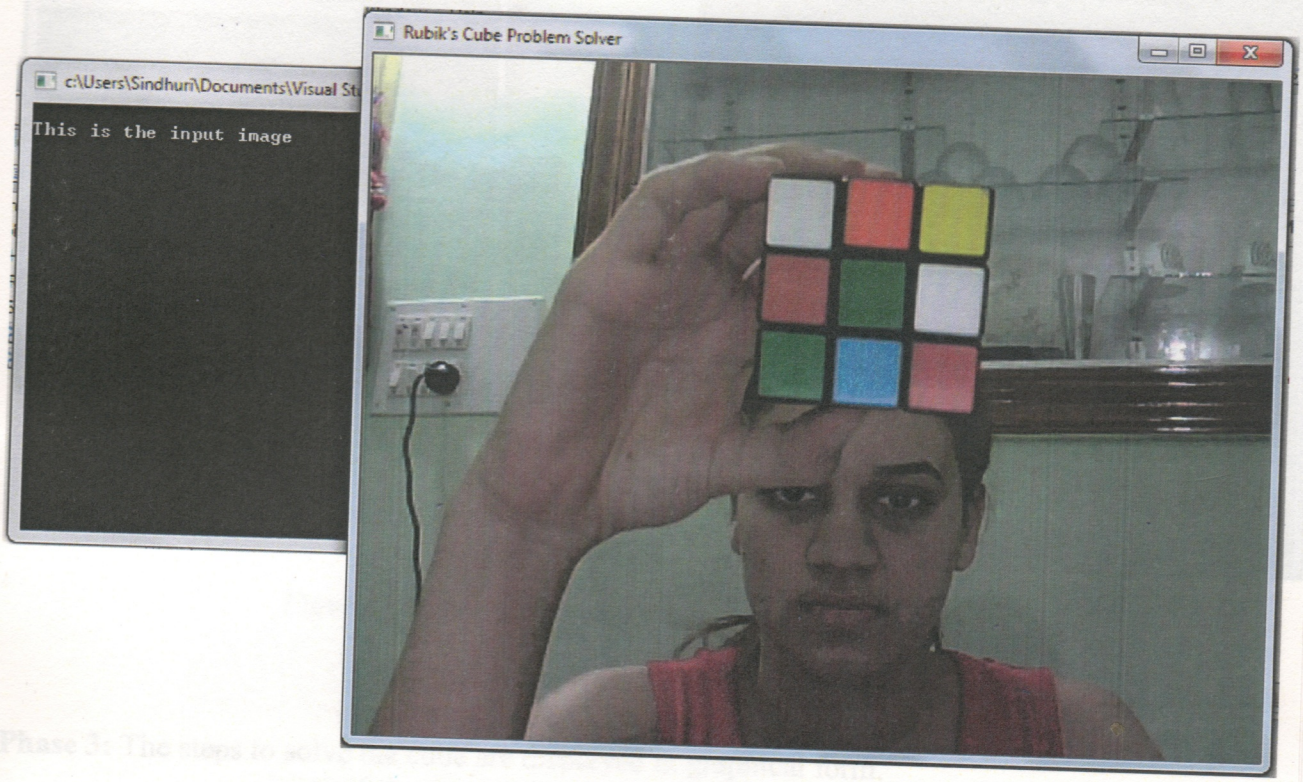

*Figure 7.4:*Input given for colour detection

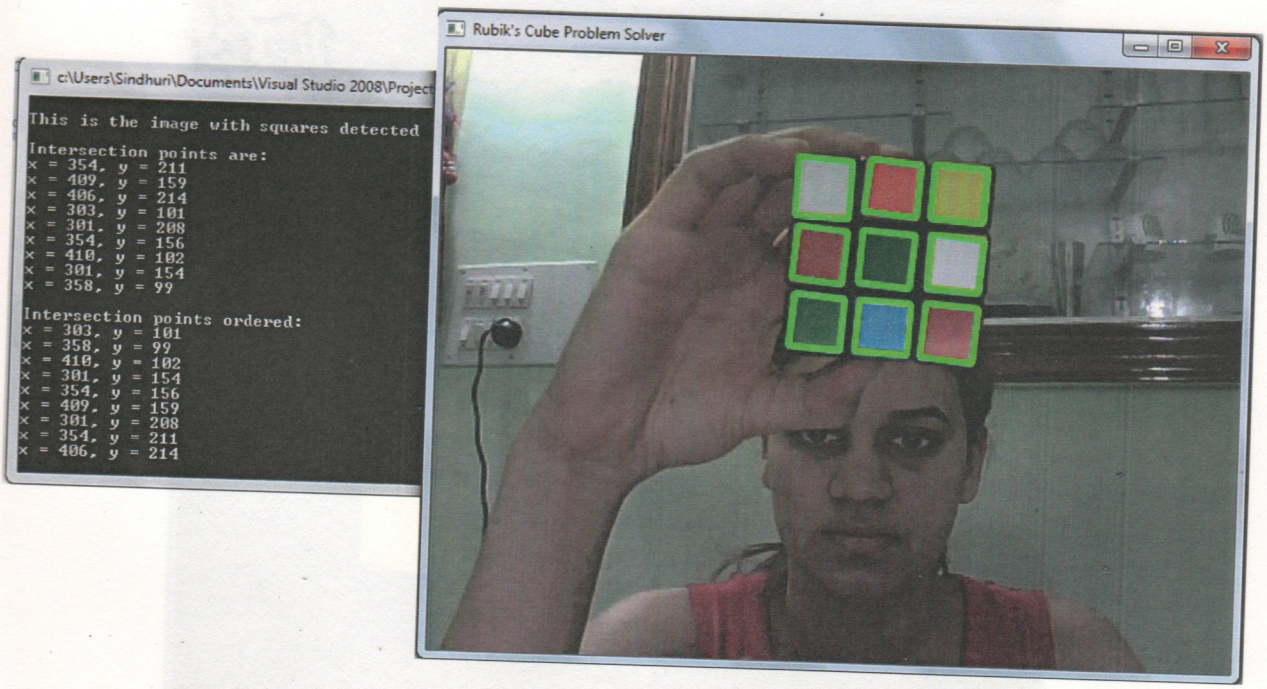

*Figure 7.5:*Output of intersection points after detection of squares

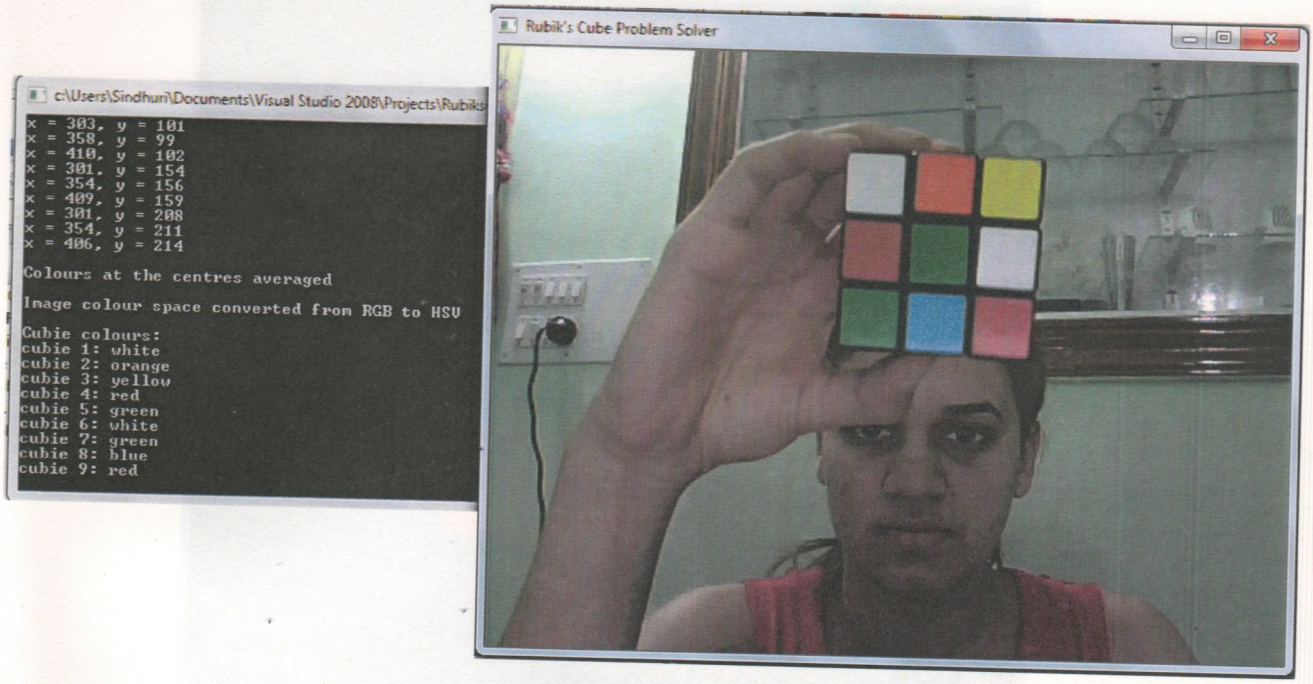

*Figure 7.6:*Output showing colours of each square

Phase 3: The steps to solve the cube are displayed in graphical form.

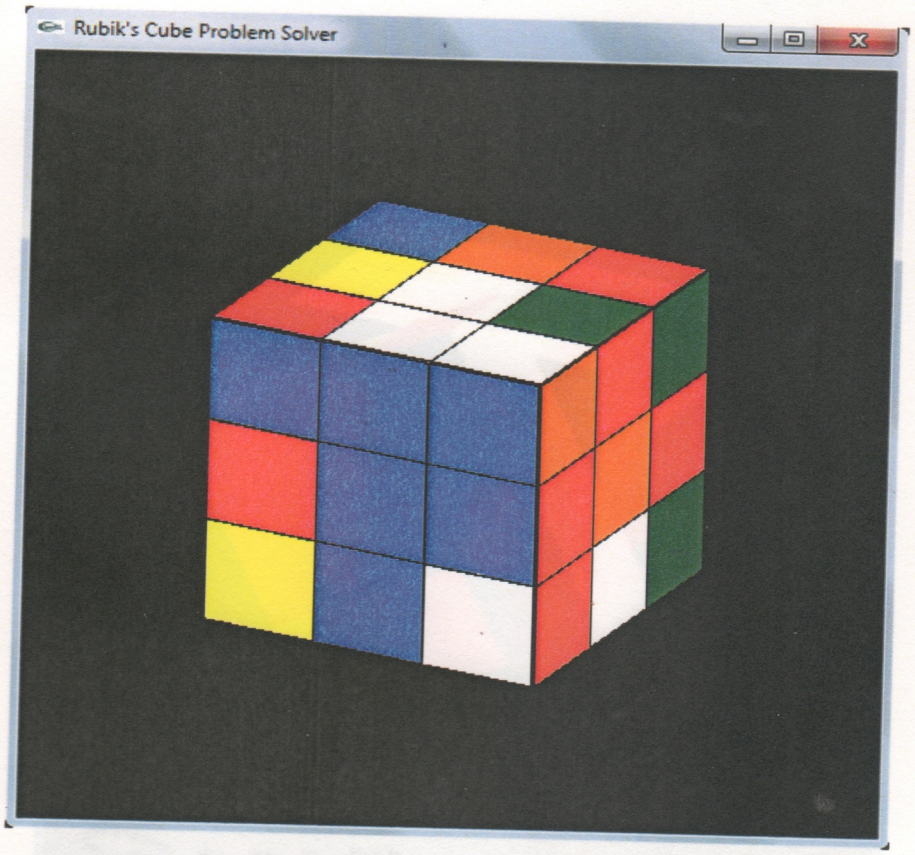

Figure 7.7: Initial Configuration of cube.

Figure 7.8 : Figure showing the rotation along x-axis

Figure 7.9 : Figure showing rotation along z-axis.

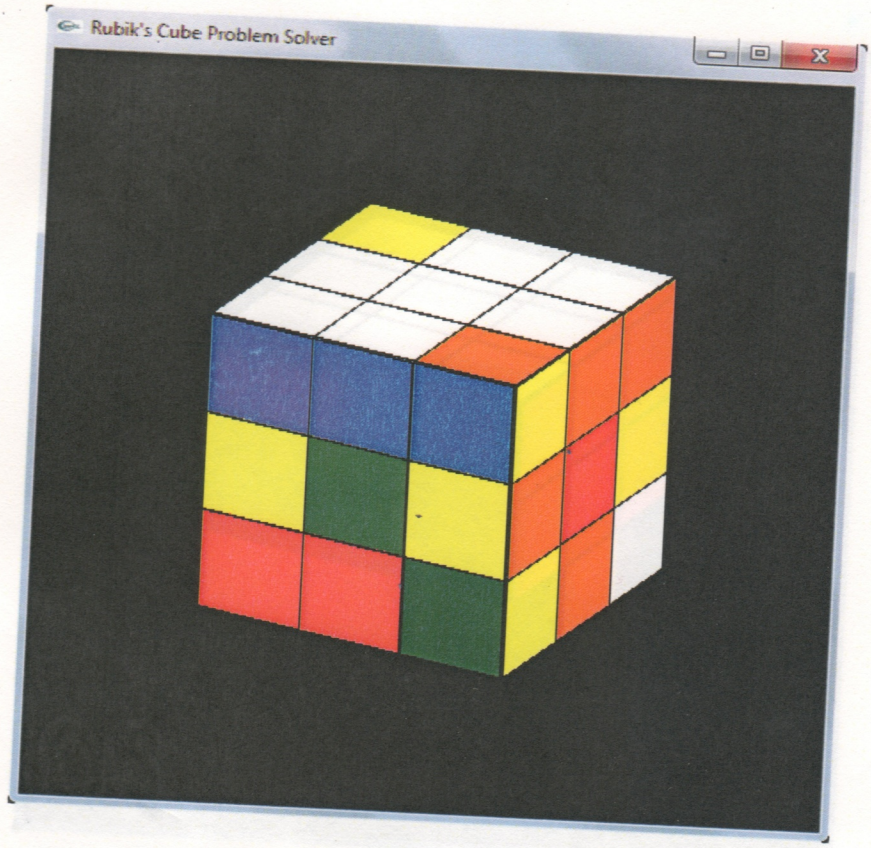

Figure 7.10: Cross formation

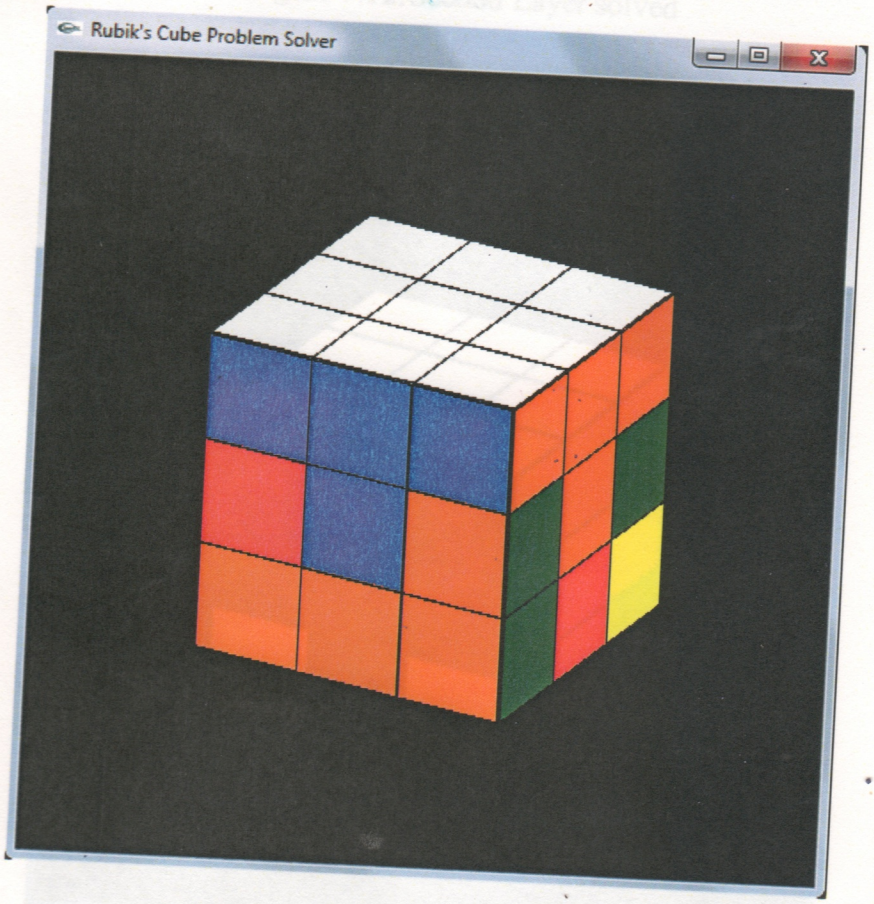

Figure 7.11 : First Layer solved.

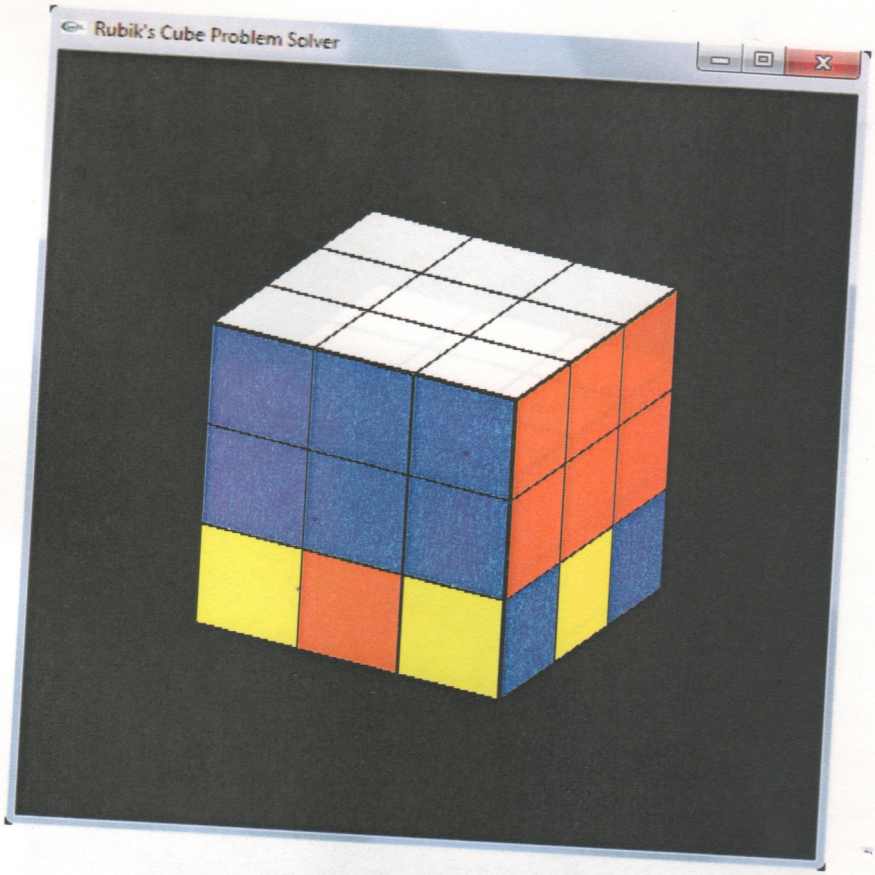

*Figure 7.12:*Second Layer solved

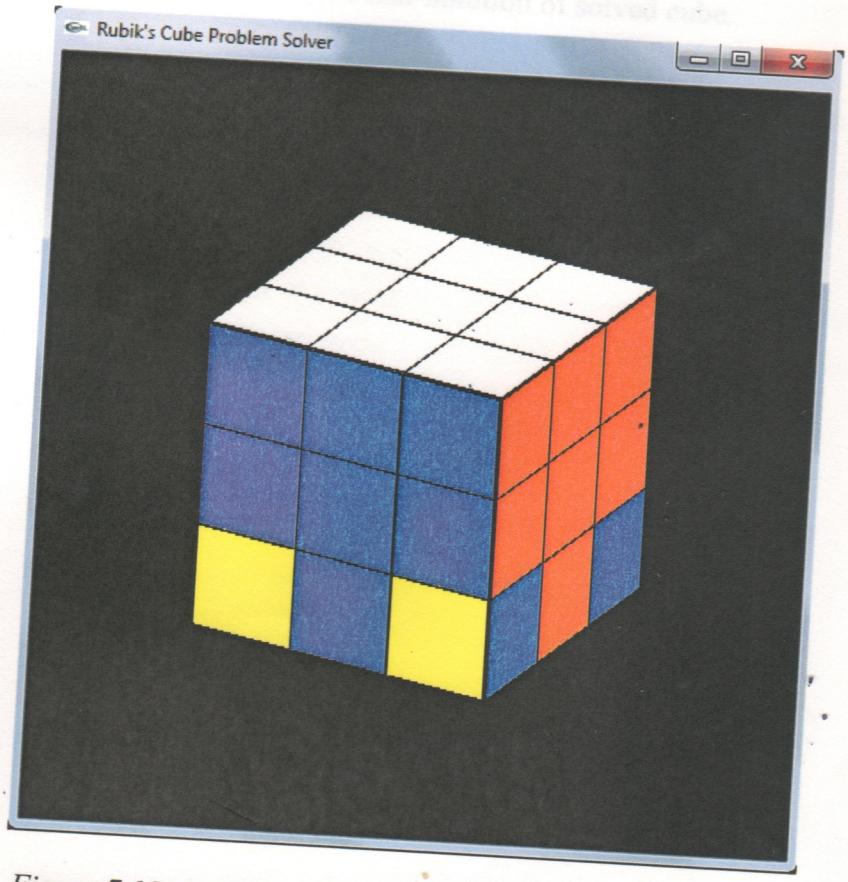

Figure 7.13 : Bottom Cross being obtained in the third layer

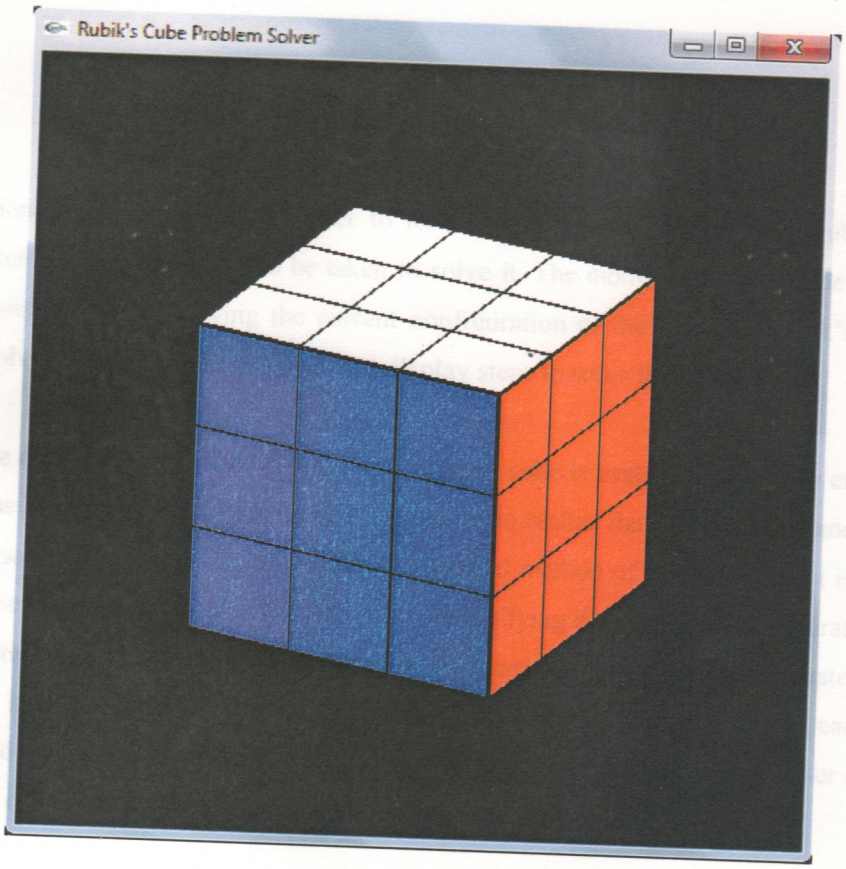

Figure 7.14: Final Solution of solved cube.

CONCLUSIONS

The application developed allows the user to let the system scan an unsolved Rubik's cubes configuration and determine the next steps to be taken to solve it. The modules have been developed by dividing into 2 sub-problems – identifying the current configuration of the scanned Rubik's cube and applying the puzzle solving algorithms to generate and display steps to solve the cube.

As on date the existing application for solving a Rubik's cube is usable by manually entering the colour of each cubie as input, after which the input is processed to realize the configuration and based on it the further steps to solve the cube is displayed. Since a manual method of providing input is followed there are chances of the input being incorrect due to which identifying the complete configuration of the cube happens to be a tough task. Hence, this Rubik's cube problem solver application facilitates to avoid input incorrectness by having made the input in the form of images - the cubes faces, with each face of the cube being captured in a frame. Compared to the manual entry of input this process is faster and easier to realize the complete configuration of the cube and provide the apt solution to solve.

Having made the work of the user easier to use the application and solve the cube. In addition to this it also serves as a learning tool for the beginners of cube solving to understand the steps to solve the cube and get a hold over certain commonly used sequential moves such as bringing a cube in particular position without disturbing the other cubes. The use of layered approach of solving highly reduces chances of incorrectness until the user makes a mistake is following the moves to solve the cube. To avoid such mistakes by the user the output of the application is given in a graphical form created using computer graphics. The end solution of the complete project including both the phases of cube configuration detection and the solving is to create a completely user friendly application helping the beginners and existing cubists.

BIBLIOGRAPHY

1. Gary Bradski and Adrian Kaehler, *Learning OpenCV*, O'Reilly, September 2008, First Edition.
2. Rafael C. Gonzalez, Richard E. Woods, Digital Image Processing, Prentice Hall,Second Edition
3. OpenGL programming guide, the official guide to learning OpenGL version 2.1,6th edition , Pearson Publications, by Dave Shreiner, Mason woo, Jackie Neider, Tom Davis.
4. http://www.discover.uottawa.ca/~qchen/my_presentations/A%20Basic%20Introduction%20to%20OpenCV%20for%20Image%20Processing.pdf
5. http://www.cs.uvm.edu/~xwu/kdd/Slides/Kmeans-ICDM06.pdf
6. http://disp.ee.ntu.edu.tw/meeting/%E6%98%B1%E7%BF%94/Segmentation%20tutorial.pdf
7. http://www.cvmt.dk/education/teaching/f10/MED8/CV/Stud/832.pdf
8. http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/Ballard__D._and_Brown__C._M.__1982__Computer_Vision.pdf
9. http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf
10. http://www.colblindor.com/color-name-hue/
11. http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html
12. http://www.autonlab.org/tutorials/kmeans.html
13. http://processing.org/learning/pixels/
14. http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html
15. http://www.aishack.in/2010/01/color-spaces/

16. http://www.aishack.in/2010/10/k-nearest-neighbors-in-opencv/
17. http://www.aishack.in/2010/08/k-means-clustering-in-opencv/
18. http://opencv.jp/opencv-2.1_org/c/miscellaneous_image_transformations.html
19. http://opencv.willowgarage.com/wiki/Welcome/Introduction
20. http://opencv.willowgarage.com/wiki/VisualC%2B%2B
21. http://opencv.willowgarage.com/documentation/python/miscellaneous_image_transformations.html
22. http://opencv.willowgarage.com/documentation/cpp/imgproc_miscellaneous_image_transformations.html
23. http://www.seas.upenn.edu/~bensapp/opencvdocs/ref/opencvref_cv.htm
24. http://www710.univ-lyon1.fr/~bouakaz/OpenCV-0.9.5/docs/ref/OpenCVRef_BasicFuncs.htm
25. http://www710.univ-lyon1.fr/~bouakaz/OpenCV-0.9.5/docs/ref/OpenCVRef_ImageProcessing.htm
26. http://www.ryanheise.com/cube/fundamental_techniques.html
27. http://www.ryanheise.com/cube/theory.html
28. http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm
29. http://homepages.inf.ed.ac.uk/rbf/HIPR2/threshld.htm
30. http://en.wikipedia.org/wiki/Computer_vision
31. http://en.wikipedia.org/wiki/Microsoft_Visual_Studio_Express
32. http://en.wikipedia.org/wiki/Smoothing
33. http://en.wikipedia.org/wiki/K-means_clustering

34. http://en.wikipedia.org/wiki/Image_noise

35. http://opencv.willowgarage.com/wiki/ObjectDetection

36. http://dasl.mem.drexel.edu/~noahKuntz/openCVTut5.html

37. http://www.cs.cmu.edu/~cil/vision.html

38. http://en.wikipedia.org/wiki/Computer_vision

39. http://en.wikipedia.org/wiki/Segmentation_%28image_processing%29

40. http://szeliski.org/Book/

41. http://www.discover.uottawa.ca/~qchen/my_presentations/A%20Basic%20Introduction%20to%20OpenCV%20for%20Image%20Processing.pdf

42. http://en.wikipedia.org/wiki/Rubik%27s_Cube

43. http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm

44. http://homepages.inf.ed.ac.uk/rbf/HIPR2/threshld.htm