Rubik's Cube Detector and Solver

By Vivek Aggala

Table Of Contents

1 Analysis

1.1 Background and Problem Areas

The Rubik's Cube is a 3-D combination puzzle invented in 1974 by Hungarian sculptor Ernő Rubik. It is one the most sold products in history and is perhaps the most famous puzzle. Yet to most people it is notoriously difficult to solve, especially at first glance. It is deceptively hard due to a number of factors. One is that each move affects the position of 8 out of 20 moving pieces or 'cubies'. Having that many pieces changing each move makes it incredibly difficult to visualise the orientation of the cube in later steps. Another problem is that there are a large number of possible positions (43 quintillion), all of which can be reached in only 20 moves[1]. That means that even a small number of turns, 7 or so, takes you deep into the state-space for the cube. Any solution needs to work carefully to reduce the number of states a rubik's cube can be in. This could be by fixing in place chosen pieces and only using pre-planned sequences of moves which keep those pieces in place, or you could restrict the types of moves which can be made. All of these factors make it very hard for beginners to learn the cube without a form of tutorial.

The most effective way to teach someone how to solve a rubik's cube has often been to provide a model of the rubik's cube to accompany the sequence of rotations needed to solve it. This isn't just true for Rubik's cube tutorials, studies have shown that visual learning is one of the most constructive ways to teach students, [2] especially if what is being taught requires a high level of analytical thinking, which is needed if you want to solve a Rubik's cube for a first time. This would include carefully looking at the cube's current orientation, determining which strategy to implement in solving its current stage and successfully implementing the correct strategy.

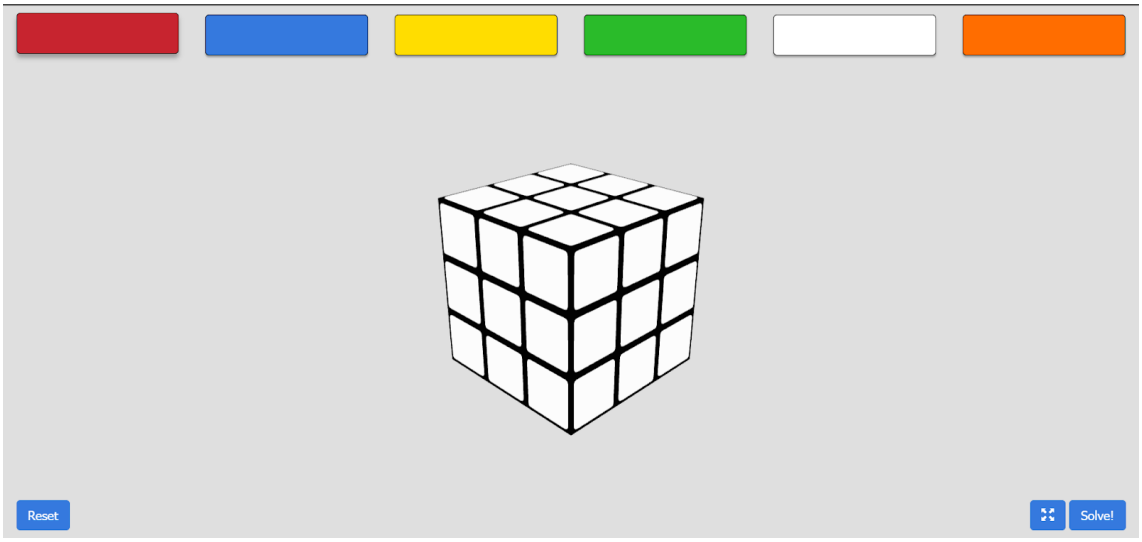
The Rubik's Cube

1.2 Proposed Solution

My solution is to create a webcam-based Rubik's cube detector and solver in python. Instead of typing in the orientation of the cube into the computer, the user can hold the cube to the computer's webcam and it will recognise the different colours on all sides of the cube. It will then implement an algorithm to solve the jumbled cube (more information on this is in the critical path section). The program also needs a way to display the correct solution for the cube. The most engaging approach to the user (and the one which I will be pursuing) is showing a three dimensional model of the cube in the orientation that the user has entered and rotating that cube in the correct moves of the cube-solving algorithm.

1.3 Description of current solutions

There are a few solutions that partially solve my problem. These solutions were researched and I was able to observe their advantages and disadvantages.
One of these was https://www.grubiks.com/solvers



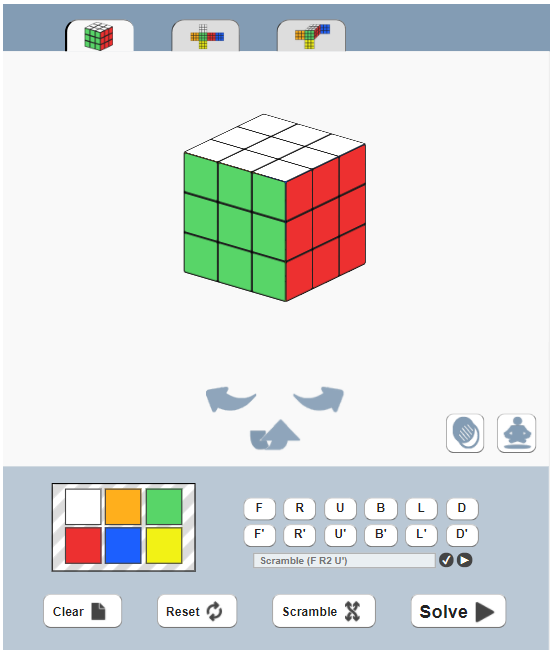
The User Interface of the grubiks cube solver

To input the scrambled permutation of the cube, the user has to manually input the colour of each small square of each face, by selecting one of the colours located at the top of the screen. Some of the advantages are that the user interface is simple and concise, yet quite aesthetically pleasing. A simple yet clean user interface is also something that I will be striving for in my solution.

However the cube does not have a scramble function, so the solver is practically unusable when the user does not have their cube at hand when the program is run. It

can also take a very long time to enter the jumbled starting orientation of the cube, as the user would have to enter 54 colours each use. This would be a feature that I would include in my implementation.

Another existing solution I found was : https://rubiks-cube-solver.com



A snapshot of the rubiks-cube-solver.com website

As the figure shows, the website is simple and straightforward to use. It has a variety of functions in which you can change the colour of each face of each cubie, rotate the cube manually and has a solving function in which it solves the cube whilst displaying each step in cube notation. It also has the option of changing the 3-D model into a net diagram of the cube, which could help the user with his understanding of the cube. Some disadvantages of this implementation is that it can take a long time in entering the jumbled starting orientation of the cube. The way that the colours are entered is, the user has to manually click on each small square and set its colour. In my implementation I will have a much faster and less tedious way of entering these colours in.

1.4 Identifying End user(s)

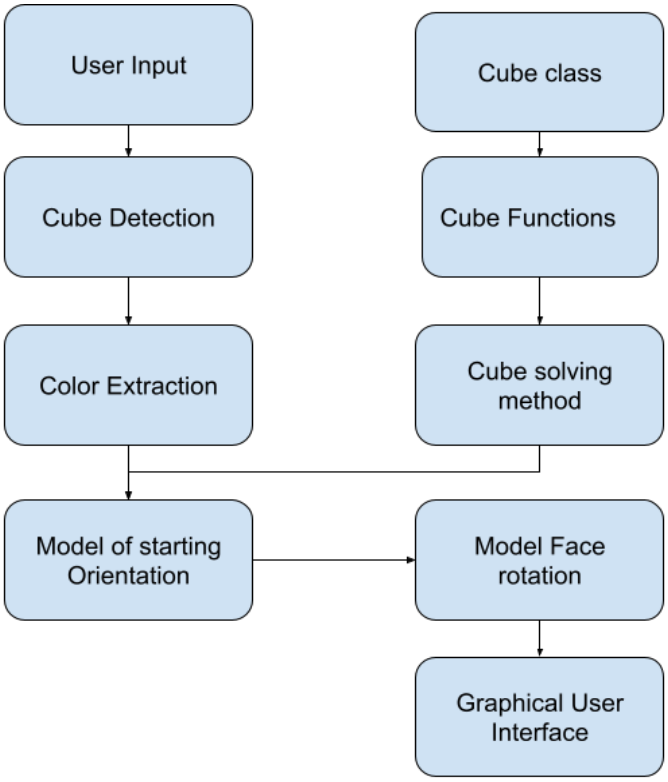
My target demographic are people that are beginners in solving a rubik's cube or are first-time solvers that engage more in visual-based learning. I have identified an end user- Dhruv Pattem. He is a first-time solver of a Rubik's cube and is interested in

learning how to solve it and fits my target demographic perfectly. I had a detailed discussion with him about what features he would like to see in my project. His main points from the conversation were:

- The program has to give clear instructions to me on which step to take next without any ambiguity.
- The program has to take into account if the user makes a mistake and needs to rewind to the last step. This is particularly relevant to me who is a beginner to cube solving as I am especially likely to take missteps often.
- If I make a mistake and lose track of which step they are on, it would be helpful for the program to adapt its current program to accommodate the new orientation of the cube.
- For users that want to learn more about cube notation, it would be useful to display the cube notation for every step of the method.
- It would be extremely helpful as a user to have a feature to show what stage of the algorithm the solver is currently on. It would be very useful when I would be learning the solving method in the future.
- The User Interface for the solver has to be easy to use and not too complicated instructions to follow. This is especially important in your project as it will be mainly used by people who have had very little to no experience in cubing.

1.4 Critical Path

The development cycle will be largely iterative, returning to previous sections to add more functionality as the project progresses. However, certain elements are prerequisites of others so must be completed first. The development process will broadly follow the order of the processes that will be completed when the program is run. The following diagram show those elements that rely on or are relied on by others – other elements of the program can be programmed at any time.



A Critical path Diagram for the development process

1.5 Requirements Specification

1) Basic Functionality
    a) Allows user to enter cube's orientation via webcam
    b) Solves cube precisely using a defined algorithm (shown below)
    c) Displays starting orientation of the cube exactly as a 3D model
    d) Displays all rotations needed to solve the cube effectively via the 3D model
    e) The program must not crash or hang upon incorrect user input or internal error and must display appropriate error messages to the user should this occur.

2) Cube detection
    a) The program must successfully calibrate to the colours of the client's cube
    b) The program must validate the cube state that the user has inputted
    c) It must identify each small square within each cube face and the colour in each square.

d) The program must be able to run on any modern laptop with either an internal or external webcam.

3) Solving Algorithm
    a) The algorithm must be of suitable difficulty to an absolute beginner to the rubik's cube.
    b) Must state each stage of the algorithm and print which stage algorithm is currently in.
    c) Program must display every move in cube notation for the user.

4) 3D Cube model
    a) The model must accurately display the faces of the cube which have been recognised from the cube detection program.
    b) The model must be able to rotate its faces by performing a rotation animation on the model
    c) The model must successfully perform the correct the correct rotations for each step of the algorithm

1.6 Requirements Specification Justification

1) Basic Functionality
    a) Entering the cube's orientation via webcam makes it quicker and more engaging than manually entering the details of each face when the program starts
    b) Using one defined method makes sure that the user is able to learn how to use this method without needing the solver in the future.
    c) Displaying the cube as a 3-D model as opposed to a 2-D model makes it even easier for the user to visualise each step of the method.
    d) For each rotation the user can orient their cube to the cube shown on screen and easily follow the instruction shown.
    e) Limiting crashes and hanging is essential to a good user experience and informing users of why the app failed allows them to either fix the problem or get more valuable support.

2) Cube Detection
    a) For different Rubik's cubes the set of 6 colours could be different so the program should be able to adapt to the user's set of colours on their cube.

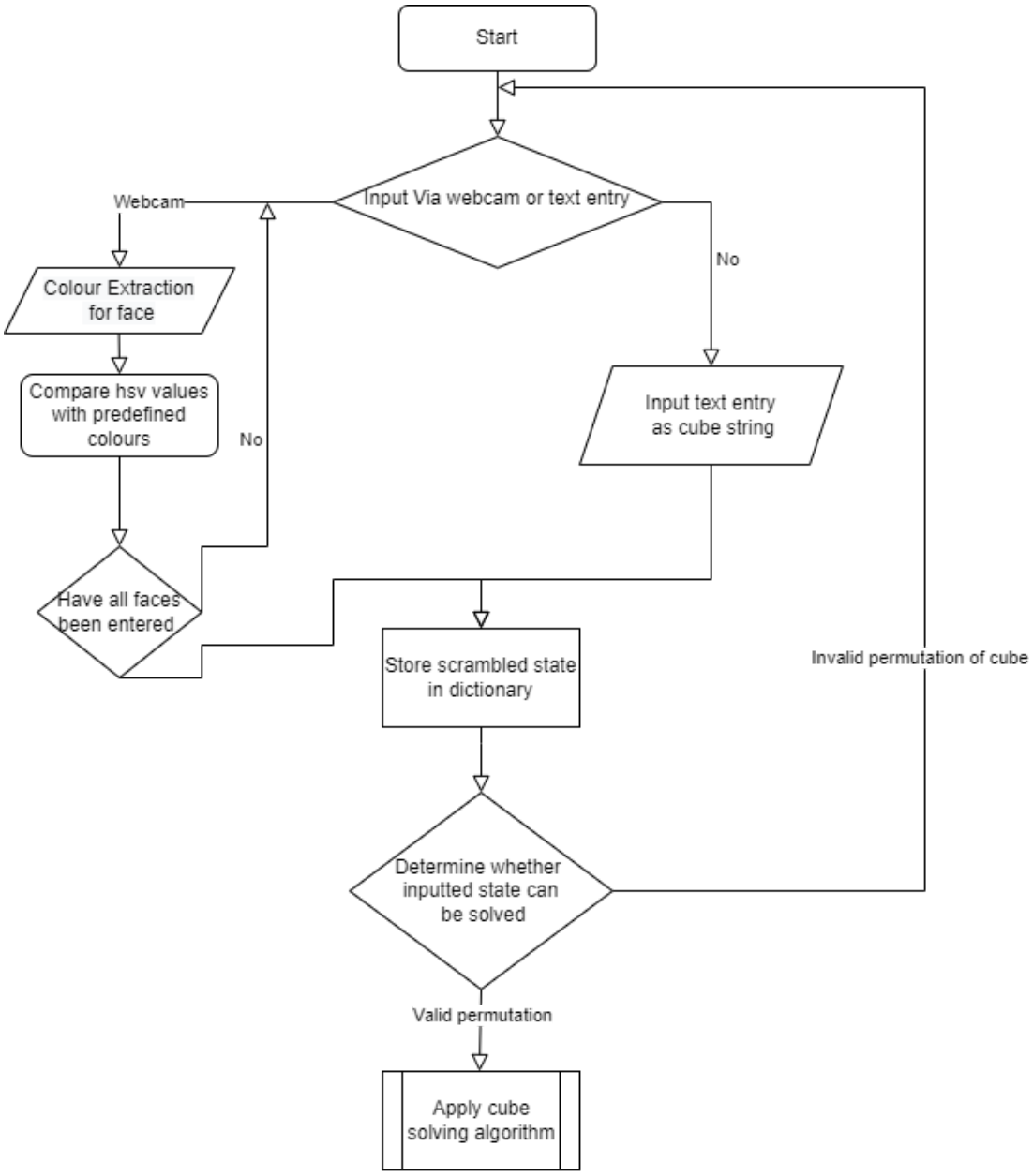
b) Validating the cube's permutation is key as the solving method will produce an error if the state is invalid.
c) Finding the colours of each small square is pivotal to the whole program as the orientation of the cube can be determined by knowing the colour of each small square and its position.
d) The program has to be appropriate to as large an audience as possible, so it must be able to be used on a computer with any webcam.

3) Solving Algorithm
   a) The difficulty of the solving method determines how fast the user is able to learn the steps of the algorithm and so for beginners, the method should be the simplest.
   b) For the user to understand and have a good chance of learning each algorithm, they need to understand the purpose of every move and to have this understanding they need to know which stage the cube is currently in.
   c) Cube notation is widely used in different solving methods and algorithms, so the user will be learning the notation to use when he ultimately looks to use more efficient and complicated algorithms.
4) 3-D Cube Model
   a) The most easy to read and informative representation of the cube's colours is in the form of the 3-D Model. Hence it seems the most appropriate way to display the colours gathered by the cube recognition.
   b) A rotation animation is quite simple to do and will mimic the user physically rotating their cube along with each instruction.
   c) Having each step as a rotation animation means that the user is able to visually follow each step from the model's rotation.

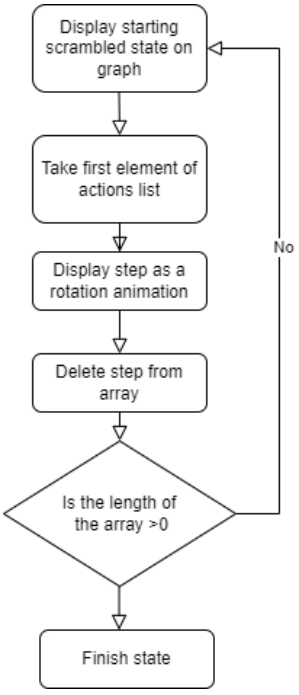
2 Documented Design

2.1 Overview

There are two major parts of the whole system (as made clear in the specification). One of which is the cube recognition software and the other is the implementation and depiction of the solving algorithm. The flow charts below show how these parts connect together to form the larger project.



A flowchart diagram detailing the process of inputting the cube

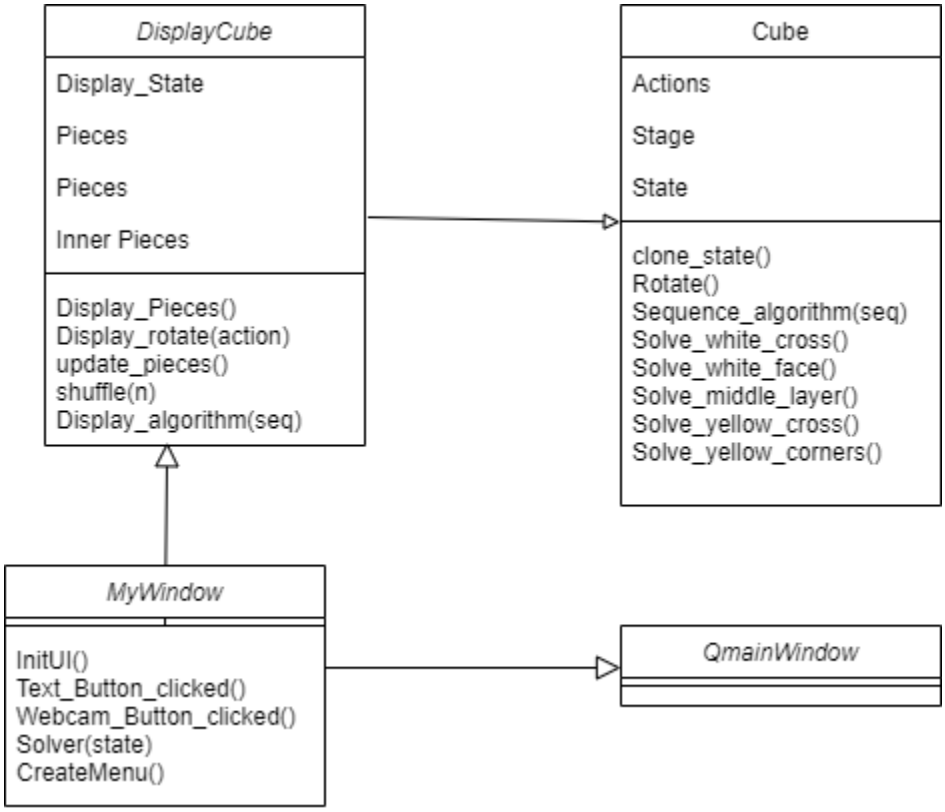


A flowchart diagram detailing the process of displaying the cube model

2.2 Data Structures

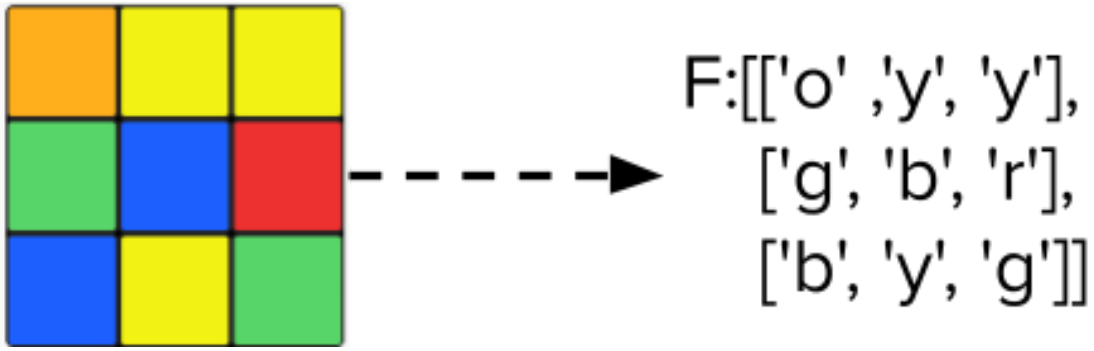
2.2.1 OOP Modelling

Object oriented programming will be used in creating the GUI for the application. However, this is very procedural, using the classes provided by PyQt5, so minimal modelling is involved. I found making two classes Cube and DisplayCube the best way to approach modelling the Rubik's cube. The cube class handles the storage of attributes like its current state. It also contains methods which implement the cube solving algorithm (explained in detail in 2.3). The DisplayCube class is a subclass of the cube class so inherits its methods and attributes. It also contains methods to display the cube state as a 3d model on Matplotlib and display all its rotations. The display cube will have to also override some of the methods in the cube class as some of the methods would need to be displayed on the model (for example the shuffle()). The figure below is a brief UML diagram to show how all these classes are connected (Qmainwindow is a class from PyQT5 which is inherited by MyWindow).



UML class diagram showing relationships between classes

The state of the cube at all times will be stored as a dictionary, with the key being the name of the face (e.g 'U') and the value being a nested list containing the colours of the squares on the face. The figure shows the mapping between the Rubik's cube face and the key value pair that is used to represent it. It is assumed that the face in the figure is a front face.



How a scrambled face would be stored in the dictionary

```
State_Initial = {'U': [['w', 'w', 'w'],['w', 'w', 'w'],['w', 'w', 'w']],
                 'D': [['y', 'y', 'y'],['y', 'y', 'y'],['y', 'y', 'y']],
                 'F': [['r', 'r', 'r'],['r', 'r', 'r'],['r', 'r', 'r']],
                 'B': [['o', 'o', 'o'],['o', 'o', 'o'],['o', 'o', 'o']],
                 'L': [['g', 'g', 'g'],['g', 'g', 'g'],['g', 'g', 'g']],
                 'R': [['b', 'b', 'b'],['b', 'b', 'b'],['b', 'b', 'b']]}
```

How a fully solved cube's state would be stored

2.2.2 Face Rotate

In order to successfully implement the cube, I needed to define a function that controls how the cube is able to rotate its faces. This is made harder by the fact that rotating different faces would need different procedures as rotating a face would involve swapping elements of select lists by index, and for different rotations, the Indexes and lists would differ each time. For example, if you rotate the upper face, that is the same as shifting the top row of each face (apart from the upper and down)to the left. So the top row of the front face goes to the left face, the left goes to the back, the back goes to the right, and the right goes to the front.

This would be coded to be something like this:

```
state['F'][0] = [copy['R'][0][j] for j in range(3)]
state['L'][0] = [copy['F'][0][j] for j in range(3)]
state['B'][0] = [copy['L'][0][j] for j in range(3)]
state['R'][0] = [copy['B'][0][j] for j in range(3)]
```

This would have to include a copy state which would store the state of the cube before the rotation, which allows the statement to refer to the cube's previous state without altering its current state. Each assignment uses list comprehension to easily find the top row of each face.

Just using these assignments would be enough if we were just looking at the faces other than the face that is being rotated. The face which is being rotated through, keeps the same colours but changes the position of these colours. In order to map the positions of the colours on this face, I used a hashmap to map a colour's initial position to its final position. It is shown below.

```
clockwise={(0, 0): (0, 2),
           (0, 1): (1, 2),
           (0, 2): (2, 2),
           (1, 0): (0, 1),
           (1, 1): (1, 1),
           (1, 2): (2, 1),
           (2, 0): (0, 0),
           (2, 1): (1, 0),
           (2, 2): (2, 0)}
```

The values in the hash map correspond to the coordinates of each small square on a face with (0,0) being the top left and (2,2). The rotation mapping can also be shown in a diagrammatic form which can be seen below. With the arrows pointing to that squares new position in a clockwise rotation

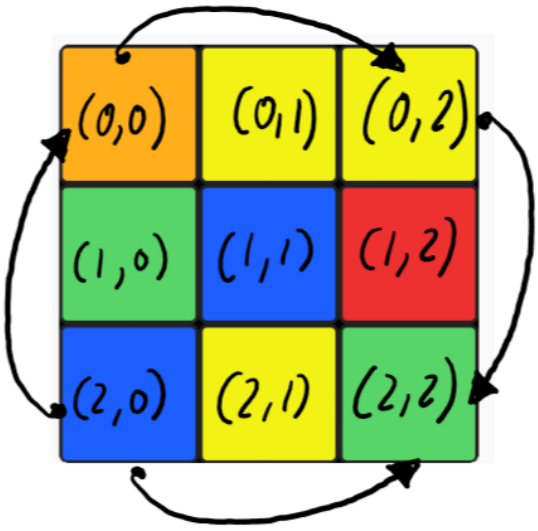


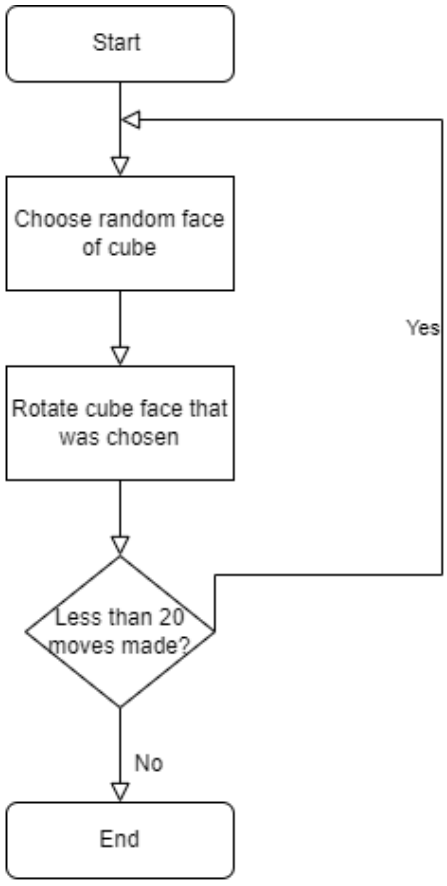
Diagram showing the mapping of each square in a rotation of a face

This would be coded as something like this, where 'face' just means whatever face is being rotated. It just swaps the colours of the coordinates of the square to the coordinates which it is mapped to in the clockwise hashmap.

```
for pos in CLOCKWISE:
                state['face'][CLOCKWISE[pos][0]][CLOCKWISE[pos][1]] = copy['face'][pos[0]][pos[1]]
```

2.2.3 Shuffle function

The cube class would also have to include a shuffle function that would be used to generate a random permutation of the cube. This would be used to test the solving method, and also could be used by the user to see the cube-solver in action without having to have a rubik's cube with them. The algorithm that I thought of for the shuffle function would start off with having an initial solved state of the cube. Then would choose a random face from the 6 faces then rotate the chosen face. This process would then be repeated n times. The value of n does not have to be excessively large, as in just as few as 20 moves there can be a possible 43 quintillion possible permutations of the cube.



A flowchart diagram detailing the shuffle function

2.3 Solving Algorithm

The solving method used to solve the cube has to be suited to beginners to cube-solving. This includes various different stages and different algorithms will be implemented at those stages. This method requires the least amount of algorithms learnt and so it is suited best for this project. Some key pieces of information when solving the cube is that the centre pieces do not move and that the cube is not solved face by face but layer by layer.

Cube Notation: A single letter denotes turning the specified face 90 degrees clockwise(U- up, D-Down, F-Front, L-Left R-Right). A letter followed by a prime (') symbol denotes a 90 degree turn anti-clockwise. If a 2 is found after a letter it means to rotate the specified face twice.

Stage 1 :The White Cross
The 1st step is to correctly position the 4 white edge pieces around the white centrepiece to create a cross. when this is made, it has to be made sure that the non-white stickers of the edge pieces also line up with their corresponding center pieces. The solving method would solve this by finding the position of all white cross pieces(edge piece) in the incorrect position and then one by one, positioning each improper edge piece to the position that it directly opposite its position in the cross and then doing a sequence (U U)

A completed white cross

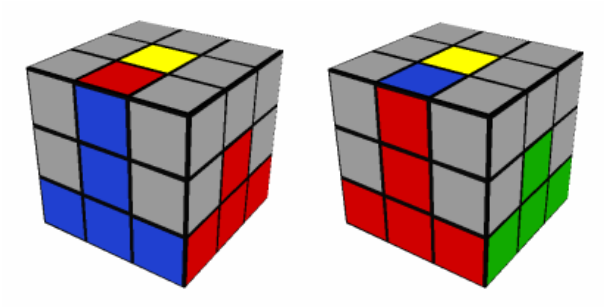
Stage 2: A full white face
This stage would involve orienting the white corner pieces to the position which not only completes the white face but also completes the bottom layer fully. A number of short sequences can be used to achieve this. If a white corner is on the bottom of an adjacent side, the bottom face is turned until the colour next to the white is the same

as its face's centre piece as shown in the figure below. Then the white corner is inserted into the white face by (R'D'R) or (FDF') depending on whether the other colour is on the left or right side of the white corner. If the corner is on the bottom face, the bottom is rotated until the white corner is directly opposite its wanted position, like in figure . Then (FLD2L'F') is used to insert it into the face.

If the corner is on the bottom face, the bottom is rotated until the white corner is directly opposite its wanted position, like in figure below . Then (F L D2 L' F ') is used to insert it into the face.

Stage 3: Second Layer Completion

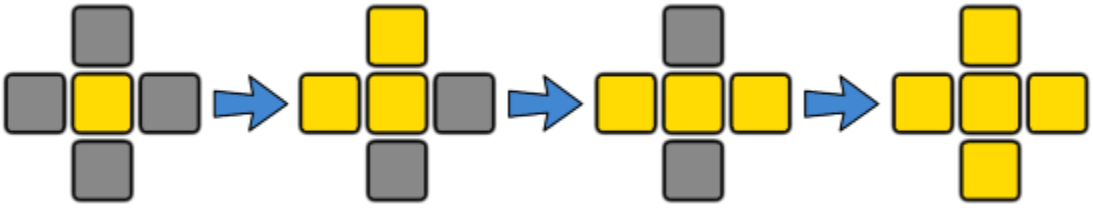
In this stage, the whole of the middle layer of the cube will be completed. There are two symmetric algorithms that will be used to make this possible. When this stage is reached the cube is flipped upside down. Next, look in the top layer (which is now the layer with the yellow face) to find one of the second layer pieces that needs to be solved. Turn the top layer until the piece forms an upside-down 'T' shape, which will be one of two possible colours depending on which way around the piece is:



As you can see in the figures above the top part of the ‘T’ shape has to either come to the right or left of the centrepiece depending on the piece's orientation. (U R U' R' U' F' U F) can be used to move the top layer edge piece to the right and (U' L' U L U F U' F') can be used to move the top layer edge piece to the left. A special case can be seen if a middle edge piece is in the correct position but its colours are in the reverse order.

Stage 4: Yellow Cross

The 4th stage’s aim is to position a yellow cross on the top face similar to the first stage’s white cross. There is only one sequence of steps (F R U R' U' F’) to be done that is repeated a number of times that is based on the starting position of yellow faces when the stage begins. The worst case scenario is a single yellow in the centre in which you would have to repeat the sequence three times, followed by an L which would take 2 sequences and then a horizontal line which would only take one. These are all shown in the figure below.



The transformation of the yellow face throughout this stage

Stage 5: Yellow Edges

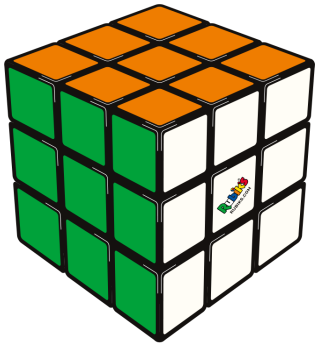
This stage involves getting the edge pieces of the yellow cube in the correct position and orientation. This can be done by repeating the sequence (R U R’ U R U2 R’), to the figure that is found below. This sequence will swap the edges that are adjacent to the yellow face until the correct permutation is achieved.

A complete yellow cross

Stage 6: Position of Yellow Corners
The aim of this stage is to place the corners in the correct position in the cube; this however does not have to be in the correct orientation. This can be done via (U R U' L' U R' U' L)

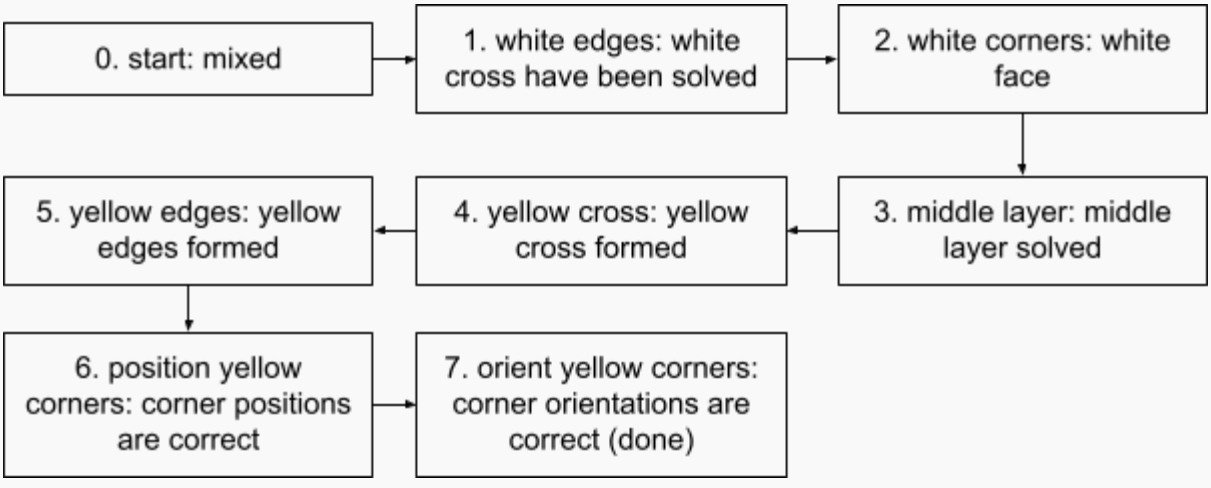
Yellow corners have been correctly positioned

Stage 7: Orientation of yellow corners
The last stage has the shortest sequence of steps (R' D' R D) but has the potential to be repeated the most times. This sequence has to be applied to all the corner pieces separately until each corner is orientated correctly. Once this is done, you will have a fully solved cube.



A fully solved rubik's cube

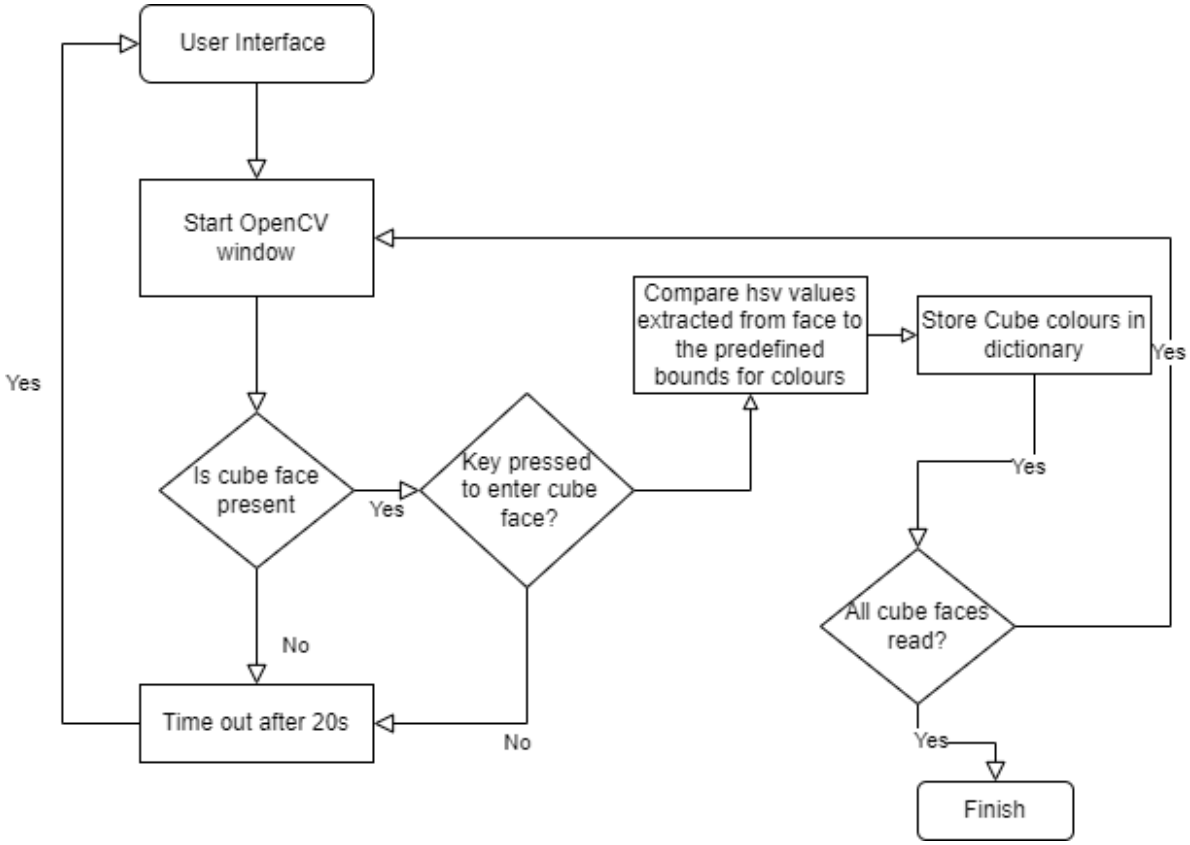
The solving algorithm and its stages can be represented by a flowchart which is shown below.



2.4 Rubik's Cube recognition

The Rubik's cube recognition software is a pivotal part of my project. It allows the user to input the scrambled permutation of their cube easily and quickly, in contrast to current implementations that require manual entry of all small squares on the cube.

A library that I will be using for the rubik's cube recognition is OpenCV. It provides a real-time optimised Computer Vision library, tools, and hardware. The sequence of the steps that the recognition software will take are represented in the flowchart below.



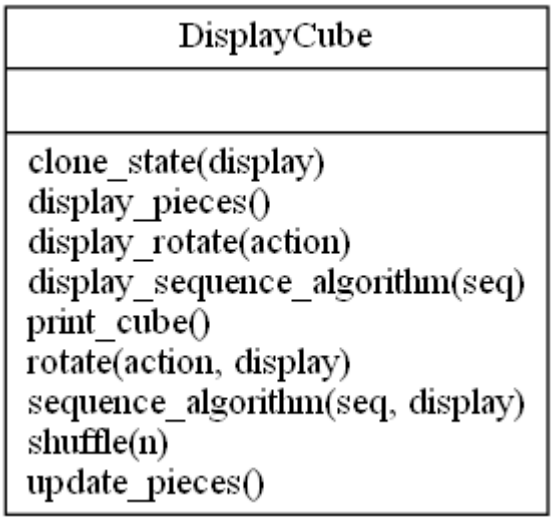
I had to also design the method of which I could input the cube's colours. I came up with the idea of creating guidelines on the video capturing window on where to hold the cube. In these areas I could therefore use OpenCV's colour extraction to return the hsv values in that region. I would then need to display what colour has been extracted. This could be in the form of displaying the colour on each cubie as text (display first letter of colour on square) as shown in the figure below.

It could also be displayed by using 6 circles which would each change colour based on the colour which has been detected (This is shown below). In the end, I used this method as it is both more aesthetically pleasing and is an easier shape to help guide where to place the cube.

The process to show which colour has been detected

2.5 Cube Model

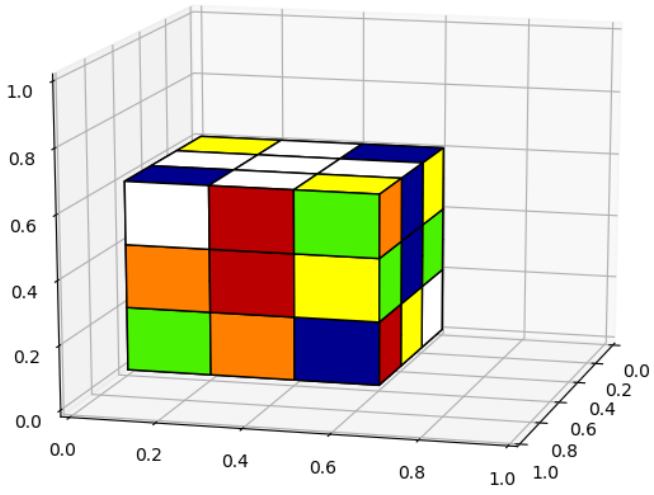
Once the starting state of the cube has been determined, it is to be displayed as a 3D model and all rotations should also be shown on that model. The display cube class which is a subclass of the cube class handles the 3d model of the cube. It's methods are shown below



Class Diagram of DisplayCube

2.5.1 Initialising the pieces

The scrambled state will be displayed on a graph as a 3D plot. Each rotation that is needed to solve the cube will be displayed as an animation changing the state of the 3D plot of the cube. In order to be able to plot the cube in 3D there is more information that the plotting software (matplotlib) would need. This would be the inner pieces behind each face of the cube. Each small cube or ‘cubie’ would need 4 separate coordinates to represent its state and it would need another 4 to represent the inner-pieces of that cubie.



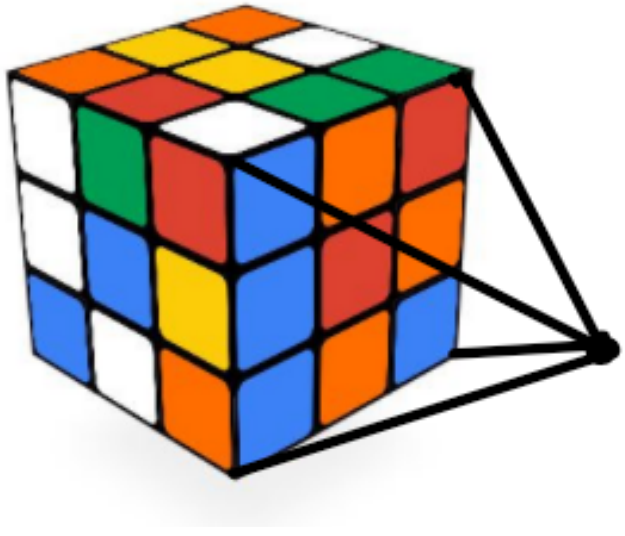
Starting state of the cube

2.5.2 Displaying a Rotation

In order to display a rotation, which would change the state of the cube on the graph, I would need to use a rotation matrix to map each x,y point to its end point when it is rotated through an angle θ. The rotation matrix is shown below for anti-clockwise rotations.

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

Unfortunately it is not as simple as just plugging in the coordinates into this matrix. Each rotation type would need a different frame of reference when rotating its face. The reference point or axis anchor would be calculated to limit the matrix to just 2 dimensions (called axis anchor as it would keep one axis constant). For example in the diagram below, as the reference point is on a plane which is parallel to that face, when the cube rotates, in that reference frame it will just be a 2D rotation which can be calculated using the rotation matrix.



A cube face with its reference point shown

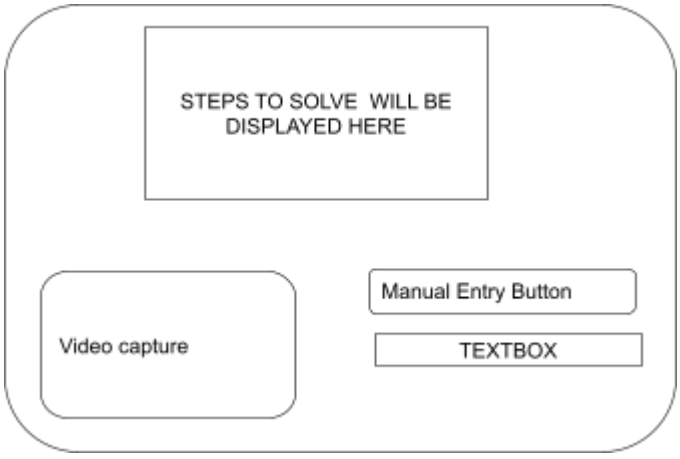
We can find the coordinates of the faces relative to the axis anchor by subtracting the absolute coordinates from the coordinates of the axis anchor. Then we can multiply the relative position vector of the coordinate by the rotation matrix and then it gives us the rotated coordinates, also in the frame of the axis anchor. Finally we would then just convert it back into absolute coordinates.

```
x_rel = x - axis_anchor
y_rel = y- axis_anchor

if clockwise :
    x_rel_new = x_rel * cos(theta) + y_rel * sin(theta)
    y_rel_new = y_rel * cos(theta) - x_rel * sin(theta)
else:

    x_rel_new = x_rel * cos(theta) - y_rel * sin(theta)
    y_rel_new = y_rel * cos(theta) + x_rel * sin(theta)

#Change relative co-ordianates to absolute co-ordinates
x_new = x_rel_new + axis_anchor[x_ax]
y_new = y_rel_new + axis_anchor[y_ax]
```

Fig Pseudocode for the change in reference frame and rotation of the points of the face

2.6 User Interface

For the User Interface I chose to use PyQT5, which is a python binding of the open-source widget-toolkit Qt, which also functions as a cross-platform application development framework. After researching existing rubik's cube solvers, it gave me an excellent understanding of what I should include in my user interface. It should be simple and concise like https://www.grubiks.com/solvers but should include all of the key functionalities of a solver like a scramble function. After planning the design of my UI, I made a few sketches that are shown below.



Version 1.1 of a sketch of the GUI

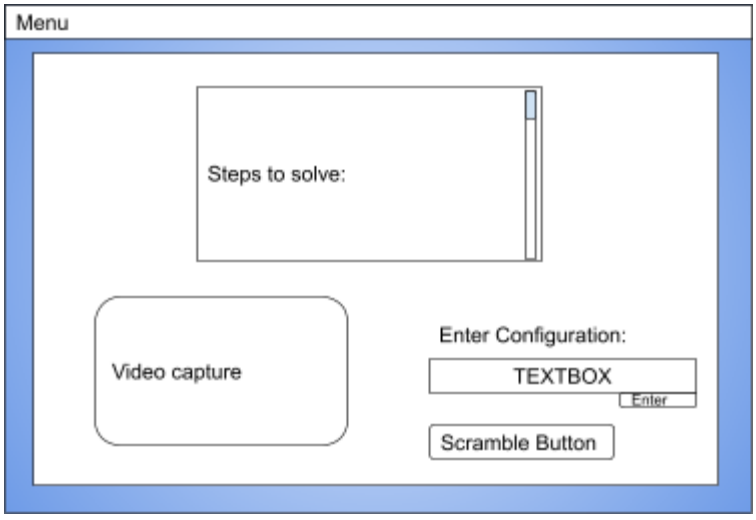
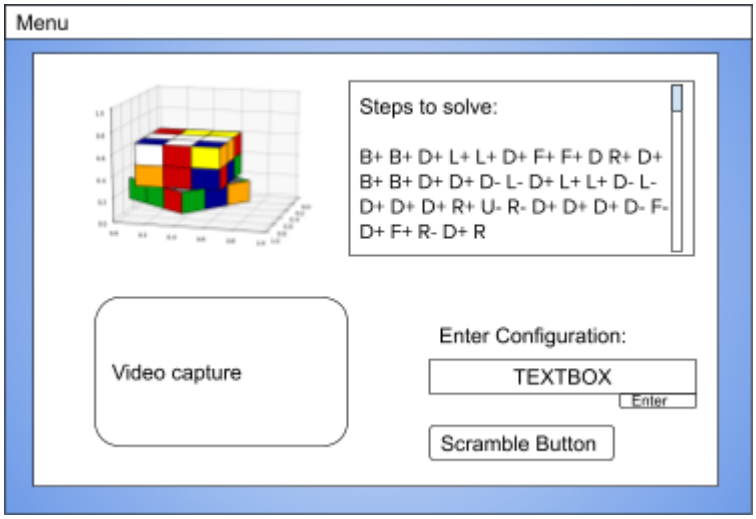
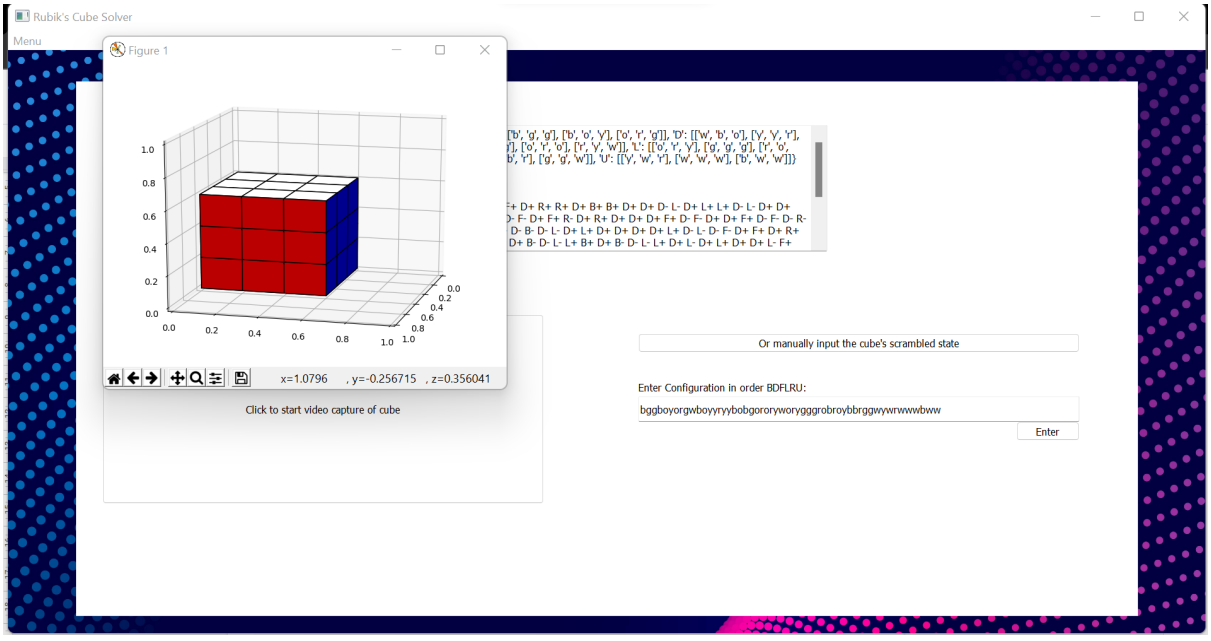


Figure Version 1.2 of UI Design

When the Video capture button is clicked, it would implement the Rubik's cube recognition detailed in Section 2.4 (creating a new window), When the video capture is completed, that window is closed and a new window is opened showing the 3D Cube model displaying all the rotations needed to solve the cube like in the figure below. There would also be the solving steps shown in cube notation in the textbox above. If instead the user opts to enter the cube's scrambled orientation manually, he can in the textbox in the bottom right corner. If this is clicked the same sequence of steps will occur, with the model appearing in a new window and the solving notation in the textbox.



The window when configuration of cube has been entered



Final design of the UI in Pyqt5

3 Technical Solution

3.1 Summary of Key Skills

Many complex algorithms were coded from scratch in the technical implementation, including the algorithms for each stage of the rubik's cube (white cross, white corners, middle layer, yellow cross, yellow edges, position yellow corners,orient yellow corners).

Advanced functions in the OpenCV library were used in the cube recognition program such as colour extraction. Efficiency of the program was also taken into consideration, using various hash-maps and dictionaries to decrease the look up time when mapping the faces to either the pieces or inner pieces that make it up. Advanced list operations such as iterating through multiple nested lists and list comprehensions were key to the manipulation of the cube's state.

Complex scientific/mathematical models: Advanced matrix operations were used in order to rotate the cube in the 3D model which were coded from scratch (rotation matrix use). Calculations of reference frames were also done when rotating a face.

A complex OOP model is used for generating graphical user interfaces and solving the cube. The displaycube class is a subclass of the cube class and inherits attributes and methods from the cube class. There is also polymorphism used in the displaycube class as it overrides functions like shuffle() which it inherits from the cube class.

3.2 Source Code

UserInterface.py[3–5]

```
#file to control interface using PyQT5
#Contains functions to connect button clicks to the cube solving method and cube model
#Textbox included to store the list of actions required to solve the cube in notation

from PyQt5 import QtWidgets
from PyQt5.QtWidgets import *
from PyQt5 import QtGui
import sys
import cube_recogntion
#from PyQt5.QtGui import *
#from PyQt5.QtCore import *
from Cube_imp import *
from cube_model import *


class MyWindow(QMainWindow):
    def __init__(self):
        #Inheritance from QMainWindow class using super()
        super(MyWindow,self).__init__()
        self.initUI()
        self._createMenu()


    def initUI(self):
        #setting window dimensions
        self.setGeometry(0,0, 1900, 955)
        self.setWindowTitle("Rubik's Cube Solver")

        #button to enter cube's state
        self.b1 = QtWidgets.QPushButton("Or manually input the cube's scrambled state",self)
        self.b1.resize(700,30)
        self.b1.move(1000,480)

        #instuctions for text entry
        self.label1 = QtWidgets.QLabel("Enter Configuration in order BDFLRU:",self)
        self.update()
        self.label1.move(1000,550)
        self.label1.resize(300, 30)
        #initialising textbox that user will use to enter scrambled state
        self.textbox = QLineEdit(self)
        self.textbox.move(1000, 580)
        self.textbox.resize(700,40)

        #Button to enter cube's permutation
        self.button =QtWidgets.QPushButton('Enter', self)
        self.button.move(1600,620)

        # connect button to function on_click
        self.button.clicked.connect(self.button_clicked1)
```

```
        #Create button that when pressed the user can enter the scrambled state via webcam
        self.b2 = QtWidgets.QPushButton(self)
        self.b2.setText("Click to start video capture of cube")
        self.b2.resize(700,300)
        #self.b2.setStyleSheet()
        self.b2.setStyleSheet(
                              "QPushButton::hover"
                              "{"
                              "border : 2px solid blue ;border-radius : 60px;"
                              "background-color : lightblue;"
                              "}")
        self.b2.clicked.connect(self.button_clicked)
        self.b2.move(150,450)

        #Initialising textbox that will display solving steps
        self.textbox1 = QPlainTextEdit(self)
        self.textbox1.move(600, 150)
        self.textbox1.resize(700,200)
        self.textbox1.insertPlainText("Steps to Solve Cube:")


# If videocam input button is clicked
    def button_clicked(self):
        self.label.setText("you pressed the button")
        state=cube_recogntion.screen_record()
        print(state)
        self.textbox1.insertPlainText(str(state)+'/n')


#If maunal text entry for permutation of cube is selected
    def button_clicked1(self):
        textboxValue = self.textbox.text()
        state={'B': [],'D': [],'F': [],'L': [],'R': [],'U': []}


        n=0
        for i in state:
            for j in range(3):
                state[i].append(list(textboxValue[n:n+3]))
                n+=3
        self.solver(state)
```

```
# inputs the scrambled state of the cube, applies the solving algorithm and displays the model
# using the DisplayCube() class

    def solver(self,state):
        textboxValue=str(state)
        self.textbox1.insertPlainText(textboxValue+'\n\n')
        cube = DisplayCube(state)
        #cube_model = DisplayCube(state)
        cube.print_cube()

        stage = cube.get_cur_stage()
        print('initial stage: ', stage)

        cube.solve()
        cube.print_cube()

        #displays steps to solve in textbox
        self.textbox1.insertPlainText("Actions:\n\n" + ' '.join(cube._actions)+'\n')


        cube.display_sequence_algorithm(cube._actions)



#menu button to help user exit the application
    def _createMenu(self):
        self.menu = self.menuBar().addMenu("&Menu")
        self.menu.addAction('&Exit', self.close)


#stylesheet of the main window which applies the background image to the UI

stylesheet = """
    MyWindow {
        border-image: url("background.png");
        background-repeat: no-repeat;
        background-position: center;
    }
"""



#Starts up the window and applies the style sheet above to the window

def window():
    app = QApplication(sys.argv)
    app.setStyleSheet(stylesheet)
    win = MyWindow()
    win.show()
    sys.exit(app.exec_())


window()
```

ColourLabeler.py[6]

```
import numpy as np
import cv2


#upper and lower bounds of the hsv values of the different colours
#This would then be compared to the hsv values in video

bounds = {
    "red" : (np.array([160, 75, 75]), np.array([180, 255, 255])),
    "blue" : (np.array([100, 75, 75]), np.array([130, 255, 255])),
    "green" : (np.array([35, 0, 0]), np.array([75, 255, 255])),
    "yellow" : (np.array([20, 75, 75]), np.array([40, 255, 255])),
    "white" : (np.array([0, 0, 20]), np.array([180, 30, 255])),
    "orange" : (np.array([10, 100, 100]), np.array([20, 255, 255]))
}

#The density function takes the hsv value of the image and performs an image mask on it.
def density(img, color):

    lower = bounds[color][0]
    upper = bounds[color][1]

    # Apply the cv2.inrange method to create a mask
    #A pixel is set to 255 if it lies within the boundaries specified otherwise set to 0.
    #This means it returns the thresholded image
    mask = cv2.inRange(img, lower, upper)
    # Apply the mask on the image to get the density of the colour
    # or how close it is to the colour

    return np.sum(mask)/255

#cubestr changes a state that is stored in a dictionary to a long cube string
#This is displayed in the order (URFDLB)

def cubestr(data):

    ret = ""
    for i in "URFDLB":

        ret += "".join(data[i])

    for i in "URFDLB":

        ret = ret.replace(data[i][4], i)
```

Cube_recognition.py [7,8]

```
#program that carries out the cube recognition and colour extraction
#Using functions from colourlabeler.py to determine which colour is present
#All colours will then be stored in the state dictionary
import numpy as np
import cv2
import time
from colour_labels import density, cubestr



def screen_record():
    #initial state that will be filled using data dictionary to match the format
    #of cube() and displaycube()
    INITIAL = {'B': [['', '', ''],['', '', ''],['', '', '']],
           'D': [['', '', ''],['', '', ''],['', '', '']],
           'F': [['', '', ''],['', '', ''],['', '', '']],
           'L': [['', '', ''],['', '', ''],['', '', '']],
           'R': [['', '', ''],['', '', ''],['', '', '']],
           'U': [['', '', ''],['', '', ''],['', '', '']]}
    last_time = time.time()
    cv2.startWindowThread()
    # cv2.namedWindow("preview")

    #Create a VideoCapture object
    cap = cv2.VideoCapture(0)
    faces = "FUDLRB"
    idx = 0
    #empty dictionary to hold the colors of each face
    data = {
        "F" : ["", "", "", "", "", "", "", "", ""],
        "U" : ["", "", "", "", "", "", "", "", ""],
        "D" : ["", "", "", "", "", "", "", "", ""],
        "L" : ["", "", "", "", "", "", "", "", ""],
        "R" : ["", "", "", "", "", "", "", "", ""],
        "B" : ["", "", "", "", "", "", "", "", ""]
    }

    while(True):
        #Read from the videocapture object
        _, img = cap.read()
        last_time = time.time()

        #opencv function to retreive hsv value of a area of the video
        img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
```

```
        #color hashmap that relates the colour to their bgr value
        # this would be the color displayed in the CV window
        colors = {
            "red" : (0, 0, 255),
            "blue" : (255, 0, 0),
            "green" : (0, 255, 0),
            "yellow" : (0, 255, 255),
            "white" : (255, 255, 255),
            "orange" : (0, 165, 255)
        }
        #offset is the distance between centres of color extraction
        offset = 75
        z = 0
        for i in (-1, 0, 1):
            for j in (-1, 0, 1):

                #centres of circles that will be shown on screen
                px = 358 + j * offset
                py = 280 + i * offset
                maxDens = [0, "white"]
                crop = img_hsv[(py-35):(py+35), (px-35):(px+35)]
                for k in ("red", "blue", "green", "yellow", "white", "orange"):
                    #density() from colour_labele
                    d = density(crop, k)
                    if d > maxDens[0]:
                        maxDens[0] = d
                        maxDens[1] = k

                #display 6 circles that change colour to the colour of each small square

                cv2.circle(img,(px, py), 5, colors[maxDens[1]], -1)
                #
                data[faces[idx]][z] = maxDens[1][0]
                z += 1

        #if q is pressed the process is quit
        cv2.imshow(faces[idx] + ' Face', img)
        if cv2.waitKey(25) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            cap.release()
            break

        #user submits the colours for the face by pressing h
        if cv2.waitKey(25) & 0xFF == ord('h'):
            idx += 1
            if idx == len(faces):

                for i, value in data.items():
                    data[i] = (np.reshape(value,(3,3))).tolist()
                for i in INITIAL:
                    INITIAL[i]=data[i]

                cv2.destroyAllWindows()
                cap.release()

                break

    return INITIAL
```

Cube_internal.py[9] [10]

```
# rubiks cube class that includes construction of cube object and the solving algorithm
#stores state of cube as a dictionary, a solved cube
#solving algorithm comprises of seven stages
#includes functions that check that stage has been reached and functions that solves each
stage

import pprint
import random

#List of available rotations that can  be made
ACTIONS = ['U+', 'U-', 'D+', 'D-', 'F+', 'F-', 'B+', 'B-', 'L+', 'L-', 'R+', 'R-']
#solved state
Initial_State = {'U': [['w', 'w', 'w'],['w', 'w', 'w'],['w', 'w', 'w']],
           'D': [['y', 'y', 'y'],['y', 'y', 'y'],['y', 'y', 'y']],
           'F': [['r', 'r', 'r'],['r', 'r', 'r'],['r', 'r', 'r']],
           'B': [['o', 'o', 'o'],['o', 'o', 'o'],['o', 'o', 'o']],
           'L': [['g', 'g', 'g'],['g', 'g', 'g'],['g', 'g', 'g']],
           'R': [['b', 'b', 'b'],['b', 'b', 'b'],['b', 'b', 'b']]}

#dictionary used to store positions if face keeps same colours when rotating
#but face changes orientation of those colours
CLOCKWISE = {
'+': {(0, 0): (0, 2),(0, 1): (1, 2),(0, 2): (2, 2),(1, 0): (0, 1),(1, 1): (1, 1),
    (1, 2): (2, 1),(2, 0): (0, 0),(2, 1): (1, 0),(2, 2): (2, 0)},
    '-': {(0, 0): (2, 0),(0, 1): (1, 0),(0, 2): (0, 0),(1, 0): (2, 1),(1, 1): (1, 1),
    (1, 2): (0, 1),(2, 0): (2, 2),(2, 1): (1, 2),(2, 2): (0, 2)}
}
#maps which face should have which colour at the end
face_to_colour = {'F': 'r','L': 'g','B': 'o','R': 'b','U': 'w','D': 'y'}

Colour_to_face = {'r': 'F','g': 'L','o': 'B','b': 'R','w': 'U','y': 'D'}


class Cube:

    def __init__(self, state=Initial_State):

        self._state = {}

        for face in ['U', 'D', 'F', 'B', 'L', 'R']:
            self._state[face] = [[state[face][i][j] for j in range(3)] for i in range(3)]

        self._actions = []
        self._stage = 0

# prints out current state of cube and steps needed to solve cube

    def print_cube(self):
        printer = pprint.PrettyPrinter(indent=4)
        print('State: ')
        printer.pprint(self._state)
        print('Actions: ')
        print(self._actions)
```

```
#shuffling function that rotates the cube n times taking a random face to rotate
    def shuffle(self, n):

        for i in range(n):
            action = random.choice(['U', 'D', 'F', 'L', 'B', 'R']) + random.choice(['+', '-'])
            self.rotate(action)


#creates a copy of the current state of cube by creating a copy dictionary and adding all elements
of state
    def clone_state(self,display=False):

        copy = {}

        if display:
            state = self._display_state
        else:
            state = self._state

        #copies out current state of cube in copy dictionary
        for face in ['U', 'D', 'F', 'B', 'L', 'R']:
            copy[face] = [[state[face][i][j] for j in range(3)] for i in range(3)]

        return copy


#function that rotates cube by swapping elements of select face arrays
    def rotate(self, action,display=False):

        #refers to cube_model.py, if display is True, then this rotation will be displayed in
        model
        if display:
            state = self._display_state
        else:
            state = self._state

        copy = self.clone_state(display)

        if not display:
            self._actions.append(action)

        # action: an element of ACTIONS
        # for each rotation there will be different elements of the 3x3 arrays
        #Clockwise(+) and anticlockwise(-) rotations will be symmetric in nature
        #Clockwise dictionary used to map colour positions if face keeps same colours when rotating
        if action == 'U+':
            state['F'][0] = [copy['R'][0][j] for j in range(3)]
            state['L'][0] = [copy['F'][0][j] for j in range(3)]
            state['B'][0] = [copy['L'][0][j] for j in range(3)]
            state['R'][0] = [copy['B'][0][j] for j in range(3)]


            for pos in CLOCKWISE['+']:

                state['U'][CLOCKWISE['+'][pos][0]][CLOCKWISE['+'][pos][1]] =
                copy['U'][pos[0]][pos[1]]
```

```
        elif action == 'U-':
            state['F'][0] = [copy['L'][0][j] for j in range(3)]
            state['L'][0] = [copy['B'][0][j] for j in range(3)]
            state['B'][0] = [copy['R'][0][j] for j in range(3)]
            state['R'][0] = [copy['F'][0][j] for j in range(3)]

            for pos in CLOCKWISE['-']:
                state['U'][CLOCKWISE['-'][pos][0]][CLOCKWISE['-'][pos][1]] = copy['U'][pos[0]][pos[1]]


        elif action == 'D+':
            state['F'][2] = [copy['L'][2][j] for j in range(3)]
            state['L'][2] = [copy['B'][2][j] for j in range(3)]
            state['B'][2] = [copy['R'][2][j] for j in range(3)]
            state['R'][2] = [copy['F'][2][j] for j in range(3)]

            for pos in CLOCKWISE['+']:
                state['D'][CLOCKWISE['+'][pos][0]][CLOCKWISE['+'][pos][1]] = copy['D'][pos[0]][pos[1]]

        elif action == 'D-':
            state['F'][2] = [copy['R'][2][j] for j in range(3)]
            state['L'][2] = [copy['F'][2][j] for j in range(3)]
            state['B'][2] = [copy['L'][2][j] for j in range(3)]
            state['R'][2] = [copy['B'][2][j] for j in range(3)]

            for pos in CLOCKWISE['-']:
                state['D'][CLOCKWISE['-'][pos][0]][CLOCKWISE['-'][pos][1]] = copy['D'][pos[0]][pos[1]]


        elif action == 'F+':
            state['U'][2] = [copy['L'][2 - i][2] for i in range(3)]

            state['L'][0][2] = copy['D'][0][0]
            state['L'][1][2] = copy['D'][0][1]
            state['L'][2][2] = copy['D'][0][2]

            state['D'][0] = [copy['R'][2 - i][0] for i in range(3)]

            state['R'][0][0] = copy['U'][2][0]
            state['R'][1][0] = copy['U'][2][1]
            state['R'][2][0] = copy['U'][2][2]

            for pos in CLOCKWISE['+']:
                state['F'][CLOCKWISE['+'][pos][0]][CLOCKWISE['+'][pos][1]] = copy['F'][pos[0]][pos[1]]

      elif action == 'F-':
            state['U'][2] = [copy['R'][i][0] for i in range(3)]

            state['R'][0][0] = copy['D'][0][2]
            state['R'][1][0] = copy['D'][0][1]
            state['R'][2][0] = copy['D'][0][0]

            state['D'][0] = [copy['L'][i][2] for i in range(3)]

            state['L'][0][2] = copy['U'][2][2]
            state['L'][1][2] = copy['U'][2][1]
            state['L'][2][2] = copy['U'][2][0]

            for pos in CLOCKWISE['-']:
                state['F'][CLOCKWISE['-'][pos][0]][CLOCKWISE['-'][pos][1]] = copy['F'][pos[0]][pos[1]]
```

```
        elif action == 'B+':
            state['U'][0] = [copy['R'][i][2] for i in range(3)]

            state['R'][0][2] = copy['D'][2][2]
            state['R'][1][2] = copy['D'][2][1]
            state['R'][2][2] = copy['D'][2][0]

            state['D'][2] = [copy['L'][i][0] for i in range(3)]

            state['L'][0][0] = copy['U'][0][2]
            state['L'][1][0] = copy['U'][0][1]
            state['L'][2][0] = copy['U'][0][0]

            for pos in CLOCKWISE['+']:
                state['B'][CLOCKWISE['+'][pos][0]][CLOCKWISE['+'][pos][1]] =
                copy['B'][pos[0]][pos[1]]

        elif action == 'B-':
            state['U'][0] = [copy['L'][2 - i][0] for i in range(3)]

            state['L'][0][0] = copy['D'][2][0]
            state['L'][1][0] = copy['D'][2][1]
            state['L'][2][0] = copy['D'][2][2]

            state['D'][2] = [copy['R'][2 - i][2] for i in range(3)]

            state['R'][0][2] = copy['U'][0][0]
            state['R'][1][2] = copy['U'][0][1]
            state['R'][2][2] = copy['U'][0][2]

            for pos in CLOCKWISE['-']:
                state['B'][CLOCKWISE['-'][pos][0]][CLOCKWISE['-'][pos][1]] =
                copy['B'][pos[0]][pos[1]]


        elif action == 'L+':
            for i in range(3):
                state['F'][i][0] = copy['U'][i][0]
                state['U'][i][0] = copy['B'][2 - i][2]
                state['B'][i][2] = copy['D'][2 - i][0]
                state['D'][i][0] = copy['F'][i][0]

            for pos in CLOCKWISE['+']:
                state['L'][CLOCKWISE['+'][pos][0]][CLOCKWISE['+'][pos][1]] =
                copy['L'][pos[0]][pos[1]]
```

```
        elif action == 'L-':
            for i in range(3):
                state['F'][i][0] = copy['D'][i][0]
                state['D'][i][0] = copy['B'][2 - i][2]
                state['B'][i][2] = copy['U'][2 - i][0]
                state['U'][i][0] = copy['F'][i][0]

            for pos in CLOCKWISE['-']:
                state['L'][CLOCKWISE['-'][pos][0]][CLOCKWISE['-'][pos][1]] =
                copy['L'][pos[0]][pos[1]]


        elif action == 'R+':
            for i in range(3):
                state['F'][i][2] = copy['D'][i][2]
                state['D'][i][2] = copy['B'][2 - i][0]
                state['B'][i][0] = copy['U'][2 - i][2]
                state['U'][i][2] = copy['F'][i][2]

            for pos in CLOCKWISE['+']:
                state['R'][CLOCKWISE['+'][pos][0]][CLOCKWISE['+'][pos][1]] =
                copy['R'][pos[0]][pos[1]]


        elif action == 'R-':
            for i in range(3):
                state['F'][i][2] = copy['U'][i][2]
                state['U'][i][2] = copy['B'][2 - i][0]
                state['B'][i][0] = copy['D'][2 - i][2]
                state['D'][i][2] = copy['F'][i][2]

            for pos in CLOCKWISE['-']:
                state['R'][CLOCKWISE['-'][pos][0]][CLOCKWISE['-'][pos][1]] =
                copy['R'][pos[0]][pos[1]]


#takes in the list of steps needed to solve the cub and rotates the cube object by those moves

    def sequence_algorithm(self, seq):

        for action in seq:
            self.rotate(action)
```

```
        #stage:
        #   0. start: mixed
        #   1. white edges: white cross have been solved
        #   2. white corners: white face
        #   3. middle layer: middle layer solved
        #   4. yellow cross: yellow cross formed
        #   5. yellow edges: yellow edges formed
        #   6. position yellow corners: corner positions are correct
        #   7. orient yellow corners: corner orientations are correct (done)

    #Function checks if white cross is finished with its adjacent pieces also being in the
right place
    def is_stage_finish_white_edges(self):

        if not self._state['U'][0][1] == self._state['U'][1][0] == self._state['U'][2][1] ==
self._state['U'][1][2] == 'w':
            return False

        if self._state['F'][0][1] != Initial_State['F'][0][1] or \
                self._state['L'][0][1] != Initial_State['L'][0][1] or \
                self._state['B'][0][1] != Initial_State['B'][0][1] or \
                self._state['R'][0][1] != Initial_State['R'][0][1]:
            return False

        return True
    #Checks if there is a full white face and that the first layer is complete by comparing
with first layer of solved state
    def is_stage_finish_white_corners(self):

        if self._state['U'] != Initial_State['U']:
            return False

        if self._state['F'][0] != Initial_State['F'][0] or \
                self._state['L'][0] != Initial_State['L'][0] or \
                self._state['B'][0] != Initial_State['B'][0] or \
                self._state['R'][0] != Initial_State['R'][0]:
            return False

        return True

    #Checks if the middle layer is complete by comparing with middle layer of solved state
    def is_stage_finish_middle_layer(self):

        if self._state['U'] != Initial_State['U']:
            return False

        if self._state['F'][:2] != Initial_State['F'][:2] or \
                self._state['L'][:2] != Initial_State['L'][:2] or \
                self._state['B'][:2] != Initial_State['B'][:2] or \
                self._state['R'][:2] != Initial_State['R'][:2]:
            return False

        return True
```

```
#Checks if yellow cross is fully formed whilst keeping the first and middle layer intact
    def is_stage_finish_yellow_cross(self):

        if self._state['U'] != Initial_State['U']:
            return False

        if self._state['F'][:2] != Initial_State['F'][:2] or \
                self._state['L'][:2] != Initial_State['L'][:2] or \
                self._state['B'][:2] != Initial_State['B'][:2] or \
                self._state['R'][:2] != Initial_State['R'][:2]:
            return False

        if not self._state['D'][1][0] == self._state['D'][0][1] == self._state['D'][2][1] ==
self._state['D'][1][2] == 'y':
            return False

        return True
#Similar to first stage, checks if white cross is finished with its adjacent pieces also being
in the right place
    def is_stage_finish_yellow_edges(self):

        if self._state['U'] != Initial_State['U']:
            return False

        if self._state['F'][:2] != Initial_State['F'][:2] or \
                self._state['L'][:2] != Initial_State['L'][:2] or \
                self._state['B'][:2] != Initial_State['B'][:2] or \
                self._state['R'][:2] != Initial_State['R'][:2]:
            return False

        if self._state['F'][2][1] != Initial_State['F'][2][1] or \
                self._state['L'][2][1] != Initial_State['L'][2][1] or \
                self._state['B'][2][1] != Initial_State['B'][2][1] or \
                self._state['R'][2][1] != Initial_State['R'][2][1]:
            return False

        if not self._state['D'][1][0] == self._state['D'][0][1] == self._state['D'][2][1] ==
self._state['D'][1][2] == 'y':
            return False

        return True

    #Checks if yellow corners are in correct position but does not have to be in correct
orientation
    def is_stage_finish_position_yellow_corners(self):

        if self._state['U'] != Initial_State['U']:
            return False

        if self._state['F'][:2] != Initial_State['F'][:2] or \
                self._state['L'][:2] != Initial_State['L'][:2] or \
                self._state['B'][:2] != Initial_State['B'][:2] or \
                self._state['R'][:2] != Initial_State['R'][:2]:
            return False
```

```
#returns stage number (stage numbers shown above) for the cube's current state
    def get_cur_stage(self):

        # mixed

        # finish white edges
        if not self._state['U'][0][1] == self._state['U'][1][0] == self._state['U'][2][1] ==
self._state['U'][1][
            2] == 'w':
            return 0

        if self._state['F'][0][1] != face_to_colour['F'] or \
                self._state['L'][0][1] != face_to_colour['L'] or \
                self._state['B'][0][1] != face_to_colour['B'] or \
                self._state['R'][0][1] != face_to_colour['R']:
            return 0

        # finish white corners
        if self._state['U'] != Initial_State['U']:
            return 1

        if self._state['F'][0] != Initial_State['F'][0] or \
                self._state['L'][0] != Initial_State['L'][0] or \
                self._state['B'][0] != Initial_State['B'][0] or \
                self._state['R'][0] != Initial_State['R'][0]:
            return 1

        # finish middle layer
        if self._state['F'][:2] != Initial_State['F'][:2] or \
                self._state['L'][:2] != Initial_State['L'][:2] or \
                self._state['B'][:2] != Initial_State['B'][:2] or \
                self._state['R'][:2] != Initial_State['R'][:2]:
            return 2

        # finish yellow cross
        if not self._state['D'][1][0] == self._state['D'][0][1] == self._state['D'][2][1] ==
self._state['D'][1][
            2] == 'y':
            return 3

        # finish yellow edges
        if self._state['F'][2][1] != face_to_colour['F'] or \
                self._state['L'][2][1] != face_to_colour['L'] or \
                self._state['B'][2][1] != face_to_colour['B'] or \
                self._state['R'][2][1] != face_to_colour['R']:
            return 4

        # finish position yellow corners
        if not self._state['F'][2][0] in ['r', 'g', 'y'] or not self._state['F'][2][2] in
['r', 'b', 'y'] or \
                not self._state['L'][2][0] in ['g', 'o', 'y'] or not self._state['L'][2][2] in
['g', 'r', 'y'] or \
                not self._state['B'][2][0] in ['o', 'b', 'y'] or not self._state['B'][2][2] in
['o', 'g', 'y'] or \
                not self._state['R'][2][0] in ['b', 'r', 'y'] or not self._state['R'][2][2] in
['b', 'o', 'y'] or \
                not self._state['D'][0][0] in ['y', 'g', 'r'] or not self._state['D'][0][2] in
['y', 'r', 'b'] or not \
                self._state['D'][2][0] in ['y', 'g', 'o'] or not self._state['D'][2][2] in
['y', 'o', 'b']:
            return 5
```

```
        if self._state['F'][:2] != Initial_State['F'][:2] or \
                self._state['L'][:2] != Initial_State['L'][:2] or \
                self._state['B'][:2] != Initial_State['B'][:2] or \
                self._state['R'][:2] != Initial_State['R'][:2]:
            return False

        if self._state['F'][2][1] != Initial_State['F'][2][1] or \
                self._state['L'][2][1] != Initial_State['L'][2][1] or \
                self._state['B'][2][1] != Initial_State['B'][2][1] or \
                self._state['R'][2][1] != Initial_State['R'][2][1]:
            return False
        #checking if last layer has a valid permutation of colours
        if (not self._state['F'][2][0] in ['r', 'g', 'y'] or (not self._state['F'][2][2] in
['r', 'b', 'y']) or \
                (not self._state['L'][2][0] in ['g', 'o', 'y'] or not self._state['L'][2][2] in
['g', 'r', 'y']) or \
                (not self._state['B'][2][0] in ['o', 'b', 'y'] or not self._state['B'][2][2] in
['o', 'g', 'y']) or \
                (not self._state['R'][2][0] in ['b', 'r', 'y'] or not self._state['R'][2][2] in
['b', 'o', 'y']) or \
                (not self._state['D'][0][0] in ['y', 'g', 'r'] or not self._state['D'][0][2] in
['y', 'r', 'b'] or not
                self._state['D'][2][0] in ['y', 'g', 'o'] or not self._state['D'][2][2] in ['y',
'o', 'b']):
            return False

        if not self._state['D'][1][0] == self._state['D'][0][1] == self._state['D'][2][1] ==
self._state['D'][1][
            2] == 'y':
            return False

        return True

    #Easily checked by comparing with fully solved cube
    def is_stage_finish_orient_yellow_corners(self):

        # done
        return self._state == Initial_State
```

```
        # finish orient yellow corners
        if not self._state == Initial_State:
            return 6

        # done
        return 7

    # solve white edges
    def find_improper_white_edge(self):

        for face in ['D', 'F', 'L', 'B', 'R']:
            for i in range(3):
                for j in range(3):
                    if self._state[face][i][j] == 'w' and (i == 1 or j == 1) and not i == j == 1:
                        return face, i, j

        for i in range(3):
            for j in range(3):
                if not i == j == 1 and self._state['U'][i][j] == 'w' and (i == 1 or j == 1):
                    if ((i, j) == (2, 1) and self._state['F'][0][1] != 'r') or \
                            ((i, j) == (1, 0) and self._state['L'][0][1] != 'g') or \
                            ((i, j) == (0, 1) and self._state['B'][0][1] != 'o') or \
                            ((i, j) == (1, 2) and self._state['R'][0][1] != 'b'):
                        return 'U', i, j

        return None, None, None

    def solve_single_white_edge(self, face, i, j):

        # adjust the white edge to downside
        if face == 'U':

            if (i, j) == (2, 1):
                self.sequence_algorithm(['F+', 'F+'])
            elif (i, j) == (1, 0):
                self.sequence_algorithm(['L+', 'L+'])
            elif (i, j) == (0, 1):
                self.sequence_algorithm(['B+', 'B+'])
            else:
                # (i, j) = (1, 2)
                self.sequence_algorithm(['R+', 'R+'])

            # adjust the white edge to downside
        elif face != 'D':
            # F, L, B, R
            if j == 1:
                if i == 0:
                    self.rotate(face + '+')
                else:
                    # i = 2
                    self.rotate(face + '-')
```

```
            if face == 'F':
                if j == 0:
                    self.sequence_algorithm(['L+', 'D+', 'L-'])
                else:
                    # j = 2 or j = 1
                    self.sequence_algorithm(['R-', 'D+', 'R+'])
            elif face == 'L':
                if j == 0:
                    self.sequence_algorithm(['B+', 'D+', 'B-'])
                else:
                    # j = 2 or j = 1
                    self.sequence_algorithm(['F-', 'D+', 'F+'])
            elif face == 'B':
                if j == 0:
                    self.sequence_algorithm(['R+', 'D+', 'R-'])
                else:
                    # j = 2 or j = 1
                    self.sequence_algorithm(['L-', 'D+', 'L+'])
            elif face == 'R':
                if j == 0:
                    self.sequence_algorithm(['F+', 'D+', 'F-'])
                else:
                    # j = 2 or j = 1
                    self.sequence_algorithm(['B-', 'D+', 'B+'])

            if j == 1:
                if i == 0:
                    self.rotate(face + '-')
                else:
                    # i = 2
                    self.rotate(face + '+')

        face, i, j = self.find_improper_white_edge()

        if face != 'D':
            print('error')
            self.print_cube()

        pos_map = {
            (1, 0): 'L',
            (0, 1): 'F',
            (1, 2): 'R',
            (2, 1): 'B'
        }

        while self._state[pos_map[(i, j)]][2][1] != face_to_colour[pos_map[(i, j)]]:
            self.rotate('D+')
            (i, j) = CLOCKWISE['+'][(i, j)]
        # will return downside improper white edge first
        # print(face, i, j)

        action = pos_map[(i, j)] + '+'

        self.sequence_algorithm([action, action])
```

```
    def solve_white_edges(self):

        face, i, j = self.find_improper_white_edge()

        while face != None:
            # improper white edge must exist
            self.solve_single_white_edge(face, i, j)
            face, i, j = self.find_improper_white_edge()

    # solve white corners

    def find_improper_white_corner(self):

        for face in ['F', 'L', 'B', 'R']:
            for j in [0, 2]:
                if self._state[face][2][j] == 'w':
                    return face, 2, j

        for face in ['F', 'L', 'B', 'R']:
            for j in [0, 2]:
                if self._state[face][0][j] == 'w':
                    return face, 0, j

        for i in [0, 2]:
            for j in [0, 2]:
                if self._state['U'][i][j] == 'w':
                    if ((i, j) == (2, 0) and (self._state['F'][0][0] != 'r' or
self._state['L'][0][2] != 'g')) or ((i, j) == (0, 0) and (self._state['L'][0][0] != 'g' or
self._state['B'][0][2] != 'o')) or ((i, j) == (0, 2) and (self._state['B'][0][0] != 'o' or
self._state['R'][0][2] != 'b')) or ((i, j) == (2, 2) and (self._state['R'][0][0] != 'b' or
self._state['F'][0][2] != 'r')):
                        return 'U', i, j

        for i in [0, 2]:
            for j in [0, 2]:
                if self._state['D'][i][j] == 'w':
                    return 'D', i, j

        return None, None, None

    def find_improper_white_corner_u(self):

        for i in [0, 2]:
            for j in [0, 2]:
                if self._state['U'][i][j] != 'w':
                    return (i, j)

                if ((i, j) == (2, 0) and (self._state['F'][0][0] != 'r' or self._state['L']
[0][2] != 'g')) or ((i, j) == (0, 0) and (self._state['L'][0][0] != 'g' or self._state['B']
[0][2] != 'o')) or ((i, j) == (0, 2) and (self._state['B'][0][0] != 'o' or self._state['R']
[0][2] != 'b')) or ((i, j) == (2, 2) and (self._state['R'][0][0] != 'b' or self._state['F']
[0][2] != 'r')):
                    return (i, j)

        return None
```

```
    #takes in a single corner and inserts it into the white face in correct orientation
    def solve_single_white_corner(self, face, i, j):

        # print(face, i, j)

        if face in ['F', 'L', 'B', 'R'] and i == 0:
            if face == 'F':
                if j == 0:
                    self.sequence_algorithm(['L+', 'D-', 'L-'])
                else:
                    # j = 2
                    self.sequence_algorithm(['R-', 'D+', 'R+'])
            elif face == 'L':
                if j == 0:
                    self.sequence_algorithm(['B+', 'D-', 'B-'])
                else:
                    # j = 2
                    self.sequence_algorithm(['F-', 'D+', 'F+'])
            elif face == 'B':
                if j == 0:
                    self.sequence_algorithm(['R+', 'D-', 'R-'])
                else:
                    # j = 2
                    self.sequence_algorithm(['L-', 'D+', 'L+'])
            else:
                # face = 'R'
                if j == 0:
                    self.sequence_algorithm(['F+', 'D-', 'F-'])
                else:
                    # j = 2
                    self.sequence_algorithm(['B-', 'D+', 'B+'])


        elif face == 'U':
            if (i, j) == (0, 0):
                self.sequence_algorithm(['L-', 'D-', 'L+'])
            elif (i, j) == (2, 0):
                self.sequence_algorithm(['L+', 'D+', 'L-'])
            elif (i, j) == (0, 2):
                self.sequence_algorithm(['R+', 'D+', 'R-'])
            else:
                # (i, j) = (2, 2)
                self.sequence_algorithm(['R-', 'D-', 'R+'])

        elif face == 'D':
            flexible_u = self.find_improper_white_corner_u()
            pos_map = {
                (0, 0): (2, 0),
                (2, 0): (0, 0),
                (0, 2): (2, 2),
                (2, 2): (0, 2)
            }
```

```
            #pos_map maps the initial position of the white corner to the correct
            postion
            flexible_d = pos_map[flexible_u]
            while (i, j) != flexible_d:
                self.rotate('D+')
                (i, j) = CLOCKWISE['+'][(i, j)]
            if flexible_u == (0, 0):
                self.sequence_algorithm(['L-', 'D+', 'L+'])
            elif flexible_u == (2, 0):
                self.sequence_algorithm(['F-', 'D+', 'F+'])
            elif flexible_u == (2, 2):
                self.sequence_algorithm(['R-', 'D+', 'R+'])
            elif flexible_u == (0, 2):
                self.sequence_algorithm(['B-', 'D+', 'B+'])

        face, i, j = self.find_improper_white_corner()


        if i != 2 or not face in ['F', 'L', 'B', 'R']:
            print('error')
            print(i, face)
            self.print_cube()
        #maps the position shift to which face to rotate
        act_map = {
            0: ['+', '-', {
                'F': 'L',  # + '+'
                'L': 'B',
                'B': 'R',
                'R': 'F'
            }],
            2: ['-', '+', {
                'F': 'R',  # + '-'
                'R': 'B',
                'B': 'L',
                'L': 'F'
            }]
        }

        if (face == 'F' and j == 0) or (face == 'L' and j == 2):
            (i_d, j_d) = (0, 0)
        elif (face == 'L' and j == 0) or (face == 'B' and j == 2):
            (i_d, j_d) = (2, 0)
        elif (face == 'B' and j == 0) or (face == 'R' and j == 2):
            (i_d, j_d) = (2, 2)
        else:
            # (face == 'R' and j == 0) or (face == 'F' and j == 2)
            (i_d, j_d) = (0, 2)
```

```
dir_map = {
            '+': {
                'F': 'R',
                'R': 'B',
                'B': 'L',
                'L': 'F'
            },
            '-': {
                'F': 'L',
                'L': 'B',
                'B': 'R',
                'R': 'F'
            }
        }
        #dir_map maps the face which is being rotated to the end face
        #With dir_map and act_map, we can create a list of actions needed to solve
        the stage
        while self._state['D'][i_d][j_d] != face_to_colour[face]:
            self.rotate('D+')
            face = dir_map['+'][face]
            (i_d, j_d) = CLOCKWISE['+'][(i_d, j_d)]

        self.sequence_algorithm(['D' + act_map[j][0], act_map[j][2][face] +
act_map[j][0], 'D' + act_map[j][1], act_map[j][2][face] + act_map[j][1]])

    def solve_white_corners(self):

        face, i, j = self.find_improper_white_corner()
        while face != None:
            # improper white corner must exist
            self.solve_single_white_corner(face, i, j)
            face, i, j = self.find_improper_white_corner()
```

```
    # find improper and solvable middle layer
    def find_improper_middle_layer(self):

        pos_map = {
            'F': (0, 1),
            'L': (1, 0),
            'B': (2, 1),
            'R': (1, 2)
        }

        for face in ['F', 'L', 'B', 'R']:
            if self._state[face][2][1] != 'y' and self._state['D'][pos_map[face]
[0]][pos_map[face][1]] != 'y':
                return face, 'D', None

        fac_map = {
            0: {
                'F': 'L',
                'L': 'B',
                'B': 'R',
                'R': 'F'
            },
            2: {
                'F': 'R',
                'R': 'B',
                'B': 'L',
                'L': 'F'
            }
        }

        for face in ['F', 'L', 'B', 'R']:
            for j in [0, 2]:
                if self._state[face][1][j] != face_to_colour[face] or
self._state[fac_map[j][face]][1][2 - j] != face_to_colour[
                    fac_map[j][face]]:
                    return face, fac_map[j][face], j

        return None, None, None

    def solve_single_middle_layer(self, face_1, face_2, j):

        # if the improper middle layer is on the side
        if j != None:
            if j == 0:
                self.sequence_algorithm(
                    ['D+', face_2 + '+', 'D-', face_2 + '-', 'D-', face_1 + '-',
'D+', face_1 + '+'])
            else:
                # j = 2
                self.sequence_algorithm(
                    ['D-', face_2 + '-', 'D+', face_2 + '+', 'D+', face_1 + '+',
'D-', face_1 + '-'])

            face_1, face_2, j = self.find_improper_middle_layer()
            print('new cube: ')
            self.print_cube()
            print('new position: ', face_1, face_2, j)
```

```
        if j != None:
            print('error')
            self.print_cube()

        pos_map = {
            'F': (0, 1),
            'L': (1, 0),
            'B': (2, 1),
            'R': (1, 2)
        }

        fac_map = {
            '+': {
                'F': 'R',
                'R': 'B',
                'B': 'L',
                'L': 'F'
            },
            '-': {
                'F': 'L',
                'L': 'B',
                'B': 'R',
                'R': 'F'
            }
        }

        while self._state[face_1][2][1] != face_to_colour[face_1]:
            self.rotate('D+')
            face_1 = fac_map['+'][face_1]

        (i_d, j_d) = pos_map[face_1]
        face_2 = Colour_to_face[self._state['D'][i_d][j_d]]
        if self._state['D'][i_d][j_d] == face_to_colour[fac_map['+'][face_1]]:
            # downside edge matches right edge
            self.sequence_algorithm(['D-', face_2 + '-', 'D+', face_2 + '+', 'D+',
face_1 + '+', 'D-', face_1 + '-'])
        else:
            # downside edge matches left edge
            self.sequence_algorithm(['D+', face_2 + '+', 'D-', face_2 + '-', 'D-',
face_1 + '-', 'D+', face_1 + '+'])

    def solve_middle_layer(self):

        face_1, face_2, j = self.find_improper_middle_layer()
        while face_1 != None:
            # improper middle layer must exist
            # print(face_1, face_2, j)

            self.solve_single_middle_layer(face_1, face_2, j)
            face_1, face_2, j = self.find_improper_middle_layer()
```

```
def solve_yellow_cross(self):

        fac_map = {
            'F': 'L',
            'L': 'B',
            'B': 'R',
            'R': 'F'
        }

        # check whether yellow cross has been solved
        local_stage, face = self.get_cur_yellow_cross_stage()
        while face != None:
            # solve current state
            if local_stage == 0:
                # not solved but to the next stage
                self.sequence_algorithm([face + '+', fac_map[face] + '+', 'D+', fac_map[face]
+ '-', 'D-', face + '-'])
            elif local_stage == 1:
                # solved
                self.sequence_algorithm([face + '+', fac_map[face] + '+', 'D+', fac_map[face]
+ '-', 'D-', face + '-'])
                self.sequence_algorithm([face + '+', fac_map[face] + '+', 'D+', fac_map[face]
+ '-', 'D-', face + '-'])
            else:
                # local_stage = 2
                # solved
                self.sequence_algorithm([face + '+', fac_map[face] + '+', 'D+', fac_map[face]
+ '-', 'D-', face + '-'])

            local_stage, face = self.get_cur_yellow_cross_stage()


    # solve yellow edges
    def find_adjacent_solved_yellow_edges(self):

        fac_map = {
            'F': 'L',
            'L': 'B',
            'B': 'R',
            'R': 'F'
        }

        for i in range(4):
            for face in ['F', 'L', 'B', 'R']:
                if self._state[face][2][1] == face_to_colour[face] and
self._state[fac_map[face]][2][1] == face_to_colour[
                    fac_map[face]]:
                    # face, fac_map[face] are the adjacent faces
                    return face

        while self._state['F'][2][1] != face_to_colour['F']:
            self.rotate('D+')

        return None
```

```
# finding improper positioned yellow corners
    # fac_map contains corner positions of the yellow face
    # if yellow corner in incorrect position it is swapped to correct corner which is in
face_to_colour

    def find_yellow_corner_with_incorrect_position(self):

        fac_map = {
            (0, 0): [('D', 0, 0), ('F', 2, 0), ('L', 2, 2)],
            (0, 2): [('D', 0, 2), ('F', 2, 2), ('R', 2, 0)],
            (2, 0): [('D', 2, 0), ('L', 2, 0), ('B', 2, 2)],
            (2, 2): [('D', 2, 2), ('R', 2, 2), ('B', 2, 0)]
        }

        for pos in fac_map:

            cor_set = [face_to_colour[face_pos[0]] for face_pos in fac_map[pos]]
            for face_pos in fac_map[pos]:
                if not self._state[face_pos[0]][face_pos[1]][face_pos[2]] in cor_set:
                    return pos

        return None

    def find_yellow_corner_with_correct_position(self):

        fac_map = {
            (0, 0): [('D', 0, 0), ('F', 2, 0), ('L', 2, 2)],
            (0, 2): [('D', 0, 2), ('F', 2, 2), ('R', 2, 0)],
            (2, 0): [('D', 2, 0), ('L', 2, 0), ('B', 2, 2)],
            (2, 2): [('D', 2, 2), ('R', 2, 2), ('B', 2, 0)]
        }

        for pos in fac_map:
            correct = True
            cor_set = [face_to_colour[face_pos[0]] for face_pos in fac_map[pos]]
            for face_pos in fac_map[pos]:
                if not self._state[face_pos[0]][face_pos[1]][face_pos[2]] in cor_set:
                    correct = False
            if correct:
                return pos

        return None
```

```
def solve_position_yellow_corners(self):

        correct_pos = self.find_yellow_corner_with_correct_position()
        while not correct_pos:
            self.sequence_algorithm(['D+', 'L+', 'D-', 'R-', 'D+', 'L-', 'D-', 'R+'])
            correct_pos = self.find_yellow_corner_with_correct_position()

        fac_map = {
            (0, 0): ['L', 'R'],
            (0, 2): ['F', 'B'],
            (2, 2): ['R', 'L'],
            (2, 0): ['B', 'F']
        }

        actions = fac_map[correct_pos]

        incorrect_pos = self.find_yellow_corner_with_incorrect_position()
        while incorrect_pos:
            self.sequence_algorithm(
                ['D+', actions[0] + '+', 'D-', actions[1] + '-', 'D+', actions[0] +
'-', 'D-', actions[1] + '+'])
            incorrect_pos = self.find_yellow_corner_with_incorrect_position()

    # solve orient yellow corners

    def find_yellow_corner_with_incorrect_orientation(self, Initial_State=False):

        fac_map = {
            (0, 0): [('D', 0, 0), ('F', 2, 0), ('L', 2, 2)],
            (0, 2): [('D', 0, 2), ('F', 2, 2), ('R', 2, 0)],
            (2, 0): [('D', 2, 0), ('L', 2, 0), ('B', 2, 2)],
            (2, 2): [('D', 2, 2), ('R', 2, 2), ('B', 2, 0)]
        }

        for pos in fac_map:
            if Initial_State:
                for face_pos in fac_map[pos]:
                    # print(pos, face_pos)

                    if self._state[face_pos[0]][face_pos[1]][face_pos[2]] !=
face_to_colour[face_pos[0]]:
                        return pos
            else:
                if self._state['D'][pos[0]][pos[1]] != 'y':
                    return pos

        return None
```

```
def solve_position_yellow_corners(self):

        correct_pos = self.find_yellow_corner_with_correct_position()
        while not correct_pos:
            #Yellow corners alogrithm
            self.sequence_algorithm(['D+', 'L+', 'D-', 'R-', 'D+', 'L-', 'D-', 'R+'])
            correct_pos = self.find_yellow_corner_with_correct_position()

        fac_map = {
            (0, 0): ['L', 'R'],
            (0, 2): ['F', 'B'],
            (2, 2): ['R', 'L'],
            (2, 0): ['B', 'F']
        }

        actions = fac_map[correct_pos]

        incorrect_pos = self.find_yellow_corner_with_incorrect_position()
        while incorrect_pos:
            #repeating algorithm until no corners are in incorrect position
            self.sequence_algorithm(
                ['D+', actions[0] + '+', 'D-', actions[1] + '-', 'D+', actions[0] +
'-', 'D-', actions[1] + '+'])
            incorrect_pos = self.find_yellow_corner_with_incorrect_position()

    # find inoriented yellow corners
    def find_yellow_corner_with_incorrect_orientation(self, Initial_State=False):

        fac_map = {
            (0, 0): [('D', 0, 0), ('F', 2, 0), ('L', 2, 2)],
            (0, 2): [('D', 0, 2), ('F', 2, 2), ('R', 2, 0)],
            (2, 0): [('D', 2, 0), ('L', 2, 0), ('B', 2, 2)],
            (2, 2): [('D', 2, 2), ('R', 2, 2), ('B', 2, 0)]
        }

        for pos in fac_map:
            if Initial_State:
                for face_pos in fac_map[pos]:
                    #if corner is not in correct orientation
                    if self._state[face_pos[0]][face_pos[1]][face_pos[2]] !=
face_to_colour[face_pos[0]]:
                        return pos
            else:
                #if corner does not have yellow on the top
                if self._state['D'][pos[0]][pos[1]] != 'y':
                    return pos

        return None
```

```
# solve all
    #
================================================================================
    '''
        stage:
            0. start: mixed
            1. white edges: white cross have been solved
            2. white corners: white face
            3. middle layer: middle layer solved
            4. yellow cross: yellow cross formed
            5. yellow edges: yellow edges formed
            6. position yellow corners: corner positions are correct
            7. orient yellow corners: corner orientations are correct (done)
    '''

    def solve(self):

        self._actions = []

        actions = {
            0: self.solve_white_edges,
            1: self.solve_white_corners,
            2: self.solve_middle_layer,
            3: self.solve_yellow_cross,
            4: self.solve_yellow_edges,
            5: self.solve_position_yellow_corners,
            6: self.solve_orient_yellow_corners
        }

        stage = self.get_cur_stage()
        while stage != 7:
            actions[stage]()
            stage = self.get_cur_stage()
            # self.print_cube()
            print('stage: ', stage)


        # prints number of moves
        print('>>>>>>> Solved in ' + str(len(self._actions)) + ' moves <<<<<<<')
    # self.print_cube()
```

Cube_model.py [1112]

```
#python program that handles visualising the cube as a 3D model
#Displays all rotations as animations changing the state of the graph
from Cube_internal import *
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from matplotlib.collections import PatchCollection, PolyCollection
import mpl_toolkits.mplot3d as a3
import matplotlib.colors as colors
import pylab as pl
import scipy as sp

#DisplayCube is a subclass of cube
#Inherits methods like rotate() and shuffle()
class DisplayCube(Cube):

    #the colour that that will be used in the model
    _piece_colors = {
        'r': '#ba0000',
        'g': '#4af202',
        'o': '#ff8000',
        'b': '#00008f',
        'w': 'w',
        'y': 'yellow',
        'n': 'gray'

    #anchors that are used when displaying rotation in model
    }
    _face_anchors = {
        'F': [3, 1.5, 1.5],
        'L': [1.5, 0, 1.5],
        'B': [0, 1.5, 1.5],
        'R': [1.5, 3, 1.5],
        'U': [1.5, 1.5, 3],
        'D': [1.5, 1.5, 0]
    }
    _n = 8
    _time = 0.0001

    _view_vertical = 15
    _view_horizontal = 15

    #_init_() overrides the _init_ function of the cube class

    def __init__(self, state= Initial_State):

        Cube.__init__(self, state)
        self.update_pieces()

        self._display_state = {}
        for face in ['U', 'D', 'F', 'B', 'L', 'R']:
            self._display_state[face] = [[state[face][i][j] for j in range(3)]
for i in range(3)]
        print(self._pieces)
```

```
def update_pieces(self):

        self._pieces = {}
        self._inner_pieces = {}
        x, y, z = np.indices((3, 3, 3))

        # add inner pieces
        for face in ['U', 'D', 'F', 'B', 'L', 'R']:
            for i in range(3):
                for j in range(3):
                    if face == 'U':
                        self._inner_pieces['i' + face + str(i) + str(j)] = [
                            [i, j, 2.5],
                            [i, j + 1, 2.5],
                            [i + 1, j + 1, 2.5],
                            [i + 1, j, 2.5]
                        ]
                        self._inner_pieces['m' + face + str(i) + str(j)] = [
                            [i, j, 1.5],
                            [i, j + 1, 1.5],
                            [i + 1, j + 1, 1.5],
                            [i + 1, j, 1.5]
                        ]
                    elif face == 'D':
                        self._inner_pieces['i' + face + str(i) + str(j)] = [
                            [3 - i, j, 0.5],
                            [3 - i, j + 1, 0.5],
                            [2 - i, j + 1, 0.5],
                            [2 - i, j, 0.5]
                        ]
                        self._inner_pieces['m' + face + str(i) + str(j)] = [
                            [3 - i, j, 1.5],
                            [3 - i, j + 1, 1.5],
                            [2 - i, j + 1, 1.5],
                            [2 - i, j, 1.5]
                        ]
                    elif face == 'F':
                        self._inner_pieces['i' + face + str(i) + str(j)] = [
                            [2.5, j, 3 - i],
                            [2.5, j + 1, 3 - i],
                            [2.5, j + 1, 2 - i],
                            [2.5, j, 2 - i]
                        ]
                        self._inner_pieces['m' + face + str(i) + str(j)] = [
                            [1.5, j, 3 - i],
                            [1.5, j + 1, 3 - i],
                            [1.5, j + 1, 2 - i],
                            [1.5, j, 2 - i]
                        ]
```

```
                elif face == 'L':
                    self._inner_pieces['i' + face + str(i) + str(j)] = [
                        [j, 0.5, 3 - i],
                        [j + 1, 0.5, 3 - i],
                        [j + 1, 0.5, 2 - i],
                        [j, 0.5, 2 - i]
                    ]
                    self._inner_pieces['m' + face + str(i) + str(j)] = [
                        [j, 1.5, 3 - i],
                        [j + 1, 1.5, 3 - i],
                        [j + 1, 1.5, 2 - i],
                        [j, 1.5, 2 - i]
                    ]
                elif face == 'B':
                    self._inner_pieces['i' + face + str(i) + str(j)] = [
                        [0.5, 3 - j, 3 - i],
                        [0.5, 2 - j, 3 - i],
                        [0.5, 2 - j, 2 - i],
                        [0.5, 3 - j, 2 - i]
                    ]
                    self._inner_pieces['m' + face + str(i) + str(j)] = [
                        [1.5, 3 - j, 3 - i],
                        [1.5, 2 - j, 3 - i],
                        [1.5, 2 - j, 2 - i],
                        [1.5, 3 - j, 2 - i]
                    ]
                else:
                    # face = 'R'
                    self._inner_pieces['i' + face + str(i) + str(j)] = [
                        [3 - j, 2.5, 3 - i],
                        [2 - j, 2.5, 3 - i],
                        [2 - j, 2.5, 2 - i],
                        [3 - j, 2.5, 2 - i]
                    ]
                    self._inner_pieces['m' + face + str(i) + str(j)] = [
                        [3 - j, 1.5, 3 - i],
                        [2 - j, 1.5, 3 - i],
                        [2 - j, 1.5, 2 - i],
                        [3 - j, 1.5, 2 - i]
                    ]
        #inner pieces are added to the model
```

```
#Plots scrambled orientation on 3d graph using matplotlib
    #Plots both pieces and inner_pieces
    def display_pieces(self):

        ax = a3.Axes3D(pl.figure())


        ax.view_init(self._view_vertical, self._view_horizontal)

        for piece in self._inner_pieces:
            vtx = 0.2 * np.array(self._inner_pieces[piece])
            #creates a 3d polygon with given vertices
            tri = a3.art3d.Poly3DCollection([vtx])
            tri.set_color(self._piece_colors['n'])
            tri.set_edgecolor('k')
            ax.add_collection3d(tri)

        for piece in self._pieces:
            vtx = 0.2 * np.array(self._pieces[piece])
            tri = a3.art3d.Poly3DCollection([vtx])
            tri.set_color(self._piece_colors[self._display_state[piece[0]]
[int(piece[1])][int(piece[2])]])
            tri.set_edgecolor('k')
            ax.add_collection3d(tri)

        pl.show()


    def print_cube(self):
        #function prints state and actions
        printer = pprint.PrettyPrinter(indent=4)
        print('State: ')
        printer.pprint(self._state)
        print('Display State: ')
        printer.pprint(self._display_state)
        print('Actions: ')
        print(self._actions)


    #overriding shuffle function in cube class() which has same name
    #Example of polymorphism
    def shuffle(self, n):

        for i in range(n):
            action = random.choice(['U', 'D', 'F', 'L', 'B', 'R']) +
random.choice(['+', '-'])
            self.rotate(action)
            self.rotate(action, display=True)
```

```
    #Function displays a sequence of rotations on the model
    def display_sequence_algorithm(self, seq):
        #initialises graph
        ax = a3.Axes3D(pl.figure())

        ax.autoscale(enable=True, tight=None)

        #Dictionary that maps a face to the cubies that are adjacent to it
        pic_map = {
            'F': ['U20', 'U21', 'U22', 'L02', 'L12', 'L22', 'D00', 'D01', 'D02',
'R00', 'R10', 'R20'],
            'L': ['U00', 'U10', 'U20', 'B02', 'B12', 'B22', 'D20', 'D10', 'D00',
'F20', 'F10', 'F00'],
            'B': ['U00', 'U01', 'U02', 'R02', 'R12', 'R22', 'D22', 'D21', 'D20',
'L00', 'L10', 'L20'],
            'R': ['U22', 'U12', 'U02', 'F02', 'F12', 'F22', 'D02', 'D12', 'D22',
'B00', 'B10', 'B20'],
            'U': ['F00', 'F01', 'F02', 'L00', 'L01', 'L02', 'B00', 'B01', 'B02',
'R00', 'R01', 'R02'],
            'D': ['F20', 'F21', 'F22', 'L20', 'L21', 'L22', 'B20', 'B21', 'B22',
'R20', 'R21', 'R22']
        }

        #Map used to determine the relative x and y axis of each face
        axs_map = {
            'F': [1, 2], 'B': [1, 2],
            'L': [0, 2], 'R': [0, 2],
            'U': [0, 1], 'D': [0, 1]
        }

        #adds the inner pieces to the pic_map dictionary
        for face in pic_map:
            new_pieces = pic_map[face].copy()
            for piece in pic_map[face]:
                new_pieces.append('i' + piece)
                new_pieces.append('m' + piece)
            pic_map[face] = new_pieces
        #
        for face in pic_map:
            for i in range(3):
                for j in range(3):
                    pic_map[face].append(face + str(i) + str(j))
                    pic_map[face].append('i' + face + str(i) + str(j))
        #theta is the angle that is turned through in a set time period
        #As theta increases the faster the cube will rotate
        theta = np.pi / (2 * self._n)

        for action in seq:
            [x_ax, y_ax] = axs_map[action[0]]
            axis_anchor = self._face_anchors[action[0]]
            for i in range(self._n):
                ax.clear()
                #Sets view point for the graph
                ax.view_init(self._view_vertical, self._view_horizontal)
                for piece_id in pic_map[action[0]]:
                    piece = self._inner_pieces[piece_id] if (piece_id[0] in ['i',
'm']) else self._pieces[piece_id]
```

```
                for vertex in piece:
                #Relative co_ordinates are co_ordinates with respect to face anchors
                        x_rel = vertex[x_ax] - axis_anchor[x_ax]
                        y_rel = vertex[y_ax] - axis_anchor[y_ax]

                        #the new relative co_ordinates are calculated for clockwise
rotation in frame of view of anchor
                        if (action[1] == '+' and (action[0] in ['F', 'L', 'U'])) or
(action[1] == '-' and (action[0] in ['D', 'B', 'R'])):
                            x_rel_new = x_rel * np.cos(theta) + y_rel * np.sin(theta)
                            y_rel_new = y_rel * np.cos(theta) - x_rel * np.sin(theta)
                        else:
                            #anti-clockwise rotation in frame of view of anchor
                            x_rel_new = x_rel * np.cos(theta) - y_rel * np.sin(theta)
                            y_rel_new = y_rel * np.cos(theta) + x_rel * np.sin(theta)

                        #Change relative co-ordianates to absolute co-ordinates
                        vertex[x_ax] = x_rel_new + axis_anchor[x_ax]
                        vertex[y_ax] = y_rel_new + axis_anchor[y_ax]

                for piece in self._inner_pieces:
                    vtx = 0.2 * np.array(self._inner_pieces[piece])
                    tri = a3.art3d.Poly3DCollection([vtx])
                    tri.set_color(self._piece_colors['n'])
                    tri.set_edgecolor('k')
                    ax.add_collection3d(tri)
                    #adds 3d collection of inner pieces to plot

                for piece in self._pieces:
                    vtx = 0.2 * np.array(self._pieces[piece])
                    tri = a3.art3d.Poly3DCollection([vtx])
                    tri.set_color(self._piece_colors[self._display_state[piece[0]]
[int(piece[1])][int(piece[2])]])
                    tri.set_edgecolor('k')
                    ax.add_collection3d(tri)
                    #adds 3d collection of outer pieces to plot

                pl.draw()
                pl.pause(self._time)

            self.rotate(action, display=True)
            self.update_pieces()
            self.print_cube()

        pl.show()
```

```python
#test functions for the cube model
def test_display():

    cube = DisplayCube()
    cube.display_pieces()

def test_rotate():

    cube = DisplayCube()
    cube.display_rotate('R+')
    cube.display_pieces()
    cube.print_cube()

def test_sequence_algorithm():

    cube = DisplayCube()
    cube.shuffle(50)
    cube.print_cube()
    cube.sequence_algorithm(['U+', 'R+'])
    cube.display_sequence_algorithm(['U+', 'R+'])
    cube.display_pieces()

def test_3dplot_1():

    ax = a3.Axes3D(pl.figure())
    for i in range(10):
        vtx = sp.rand(3,3)
        tri = a3.art3d.Poly3DCollection([vtx])
        tri.set_color(colors.rgb2hex(sp.rand(3)))
        tri.set_edgecolor('k')
        ax.add_collection3d(tri)
    pl.show()

def test_3dplot_2():

    ax = a3.Axes3D(pl.figure())
    vtx = [
        [0, 0, 0],
        [0, 0, 1],
        [1, 0, 1],
        [1, 0, 0]
    ]
    tri = a3.art3d.Poly3DCollection([vtx])
    tri.set_color('yellow')
    tri.set_edgecolor('k')
    ax.add_collection3d(tri)
    pl.show()

def test_display_solve():

    cube = DisplayCube()
    cube.shuffle(50)
    cube.print_cube()
    stage = cube.get_cur_stage()
    #print(stage)
    cube.solve()
    cube.print_cube()
    cube.display_sequence_algorithm(cube._actions)
```

Testing.py

```
from cube_internal import Cube

#Program contains various test states that will be used to test the solving program
#Separate functions to test each stages of the solving program(Unit Testing)
STATE_0 = {
    'B': [['r', 'g', 'o'], ['w', 'o', 'o'], ['r', 'w', 'g']],
    'D': [['w', 'r', 'b'], ['o', 'y', 'o'], ['r', 'b', 'y']],
    'F': [['b', 'y', 'g'], ['y', 'r', 'r'], ['b', 'y', 'o']],
    'L': [['y', 'r', 'r'], ['b', 'g', 'o'], ['w', 'g', 'o']],
    'R': [['w', 'b', 'b'], ['b', 'b', 'g'], ['y', 'w', 'g']],
    'U': [['g', 'r', 'y'], ['w', 'w', 'y'], ['w', 'g', 'o']]
}

STATE_1 = {
    'B': [['y', 'o', 'g'], ['o', 'o', 'r'], ['r', 'y', 'g']],
    'D': [['b', 'o', 'y'], ['y', 'y', 'y'], ['w', 'g', 'w']],
    'F': [['y', 'r', 'r'], ['r', 'r', 'o'], ['o', 'y', 'b']],
    'L': [['r', 'g', 'o'], ['g', 'g', 'b'], ['o', 'r', 'w']],
    'R': [['g', 'b', 'r'], ['g', 'b', 'b'], ['o', 'b', 'b']],
    'U': [['w', 'w', 'b'], ['w', 'w', 'w'], ['g', 'w', 'y']]
}

STATE_2 = {
    'B': [['o', 'o', 'o'], ['g', 'o', 'b'], ['y', 'g', 'r']],
    'D': [['b', 'r', 'g'], ['y', 'y', 'b'], ['y', 'y', 'o']],
    'F': [['r', 'r', 'r'], ['y', 'r', 'b'], ['y', 'g', 'r']],
    'L': [['g', 'g', 'g'], ['o', 'g', 'r'], ['b', 'o', 'o']],
    'R': [['b', 'b', 'b'], ['r', 'b', 'o'], ['y', 'y', 'g']],
    'U': [['w', 'w', 'w'], ['w', 'w', 'w'], ['w', 'w', 'w']]
}

STATE_3 = {
    'B': [['o', 'o', 'o'], ['o', 'o', 'o'], ['o', 'g', 'y']],
    'D': [['r', 'b', 'y'], ['o', 'y', 'y'], ['b', 'y', 'g']],
    'F': [['r', 'r', 'r'], ['r', 'r', 'r'], ['g', 'y', 'b']],
    'L': [['g', 'g', 'g'], ['g', 'g', 'g'], ['r', 'y', 'y']],
    'R': [['b', 'b', 'b'], ['b', 'b', 'b'], ['o', 'r', 'y']],
    'U': [['w', 'w', 'w'], ['w', 'w', 'w'], ['w', 'w', 'w']]
}

STATE_4 = {
    'B': [['o', 'o', 'o'], ['o', 'o', 'o'], ['g', 'b', 'r']],
    'D': [['g', 'y', 'o'], ['y', 'y', 'y'], ['y', 'y', 'y']],
    'F': [['r', 'r', 'r'], ['r', 'r', 'r'], ['y', 'o', 'y']],
    'L': [['g', 'g', 'g'], ['g', 'g', 'g'], ['b', 'g', 'r']],
    'R': [['b', 'b', 'b'], ['b', 'b', 'b'], ['b', 'r', 'o']],
    'U': [['w', 'w', 'w'], ['w', 'w', 'w'], ['w', 'w', 'w']]
}

STATE_5 = {
    'B': [['o', 'o', 'o'], ['o', 'o', 'o'], ['g', 'o', 'y']],
    'D': [['o', 'y', 'r'], ['y', 'y', 'y'], ['o', 'y', 'r']],
    'F': [['r', 'r', 'r'], ['r', 'r', 'r'], ['y', 'r', 'b']],
    'L': [['g', 'g', 'g'], ['g', 'g', 'g'], ['b', 'g', 'g']],
    'R': [['b', 'b', 'b'], ['b', 'b', 'b'], ['y', 'b', 'y']],
    'U': [['w', 'w', 'w'], ['w', 'w', 'w'], ['w', 'w', 'w']]
}
```

```
STATE_6 = {
    'B': [['o', 'o', 'o'], ['o', 'o', 'o'], ['y', 'o', 'o']],
    'D': [['g', 'y', 'r'], ['y', 'y', 'y'], ['y', 'y', 'b']],
    'F': [['r', 'r', 'r'], ['r', 'r', 'r'], ['y', 'r', 'b']],
    'L': [['g', 'g', 'g'], ['g', 'g', 'g'], ['g', 'g', 'r']],
    'R': [['b', 'b', 'b'], ['b', 'b', 'b'], ['y', 'b', 'o']],
    'U': [['w', 'w', 'w'], ['w', 'w', 'w'], ['w', 'w', 'w']]
}


TEST_STATE = {
    'B': [['b', 'g', 'g'], ['b', 'o', 'y'], ['o', 'r', 'g']],
    'D': [['w', 'b', 'o'], ['y', 'y', 'r'], ['y', 'y', 'b']],
    'F': [['o', 'b', 'g'], ['o', 'r', 'o'], ['r', 'y', 'w']],
    'L': [['o', 'r', 'y'], ['g', 'g', 'g'], ['r', 'o', 'b']],
    'R': [['r', 'o', 'y'], ['b', 'b', 'r'], ['g', 'g', 'w']],
    'U': [['y', 'w', 'r'], ['w', 'w', 'w'], ['b', 'w', 'w']]
}

#--------------------------------------------------------------------------------
#Functions to test each stage of algorithm

def test_solve_white_edges(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 0: ', True)

    cube.solve_white_edges()

    print('stage: ', cube.get_cur_stage());
    print('confirm 1: ', cube.is_stage_finish_white_edges())

    cube.print_cube()


def test_solve_white_corners(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 1: ', cube.is_stage_finish_white_edges())

    cube.solve_white_corners()

    print('stage: ', cube.get_cur_stage());
    print('confirm 2: ', cube.is_stage_finish_white_corners())

    cube.print_cube()


def test_solve_middle_layer(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 2: ', cube.is_stage_finish_white_corners())

    cube.solve_middle_layer()

    print('stage: ', cube.get_cur_stage());
    print('confirm 3: ', cube.is_stage_finish_middle_layer())

    cube.print_cube()
```

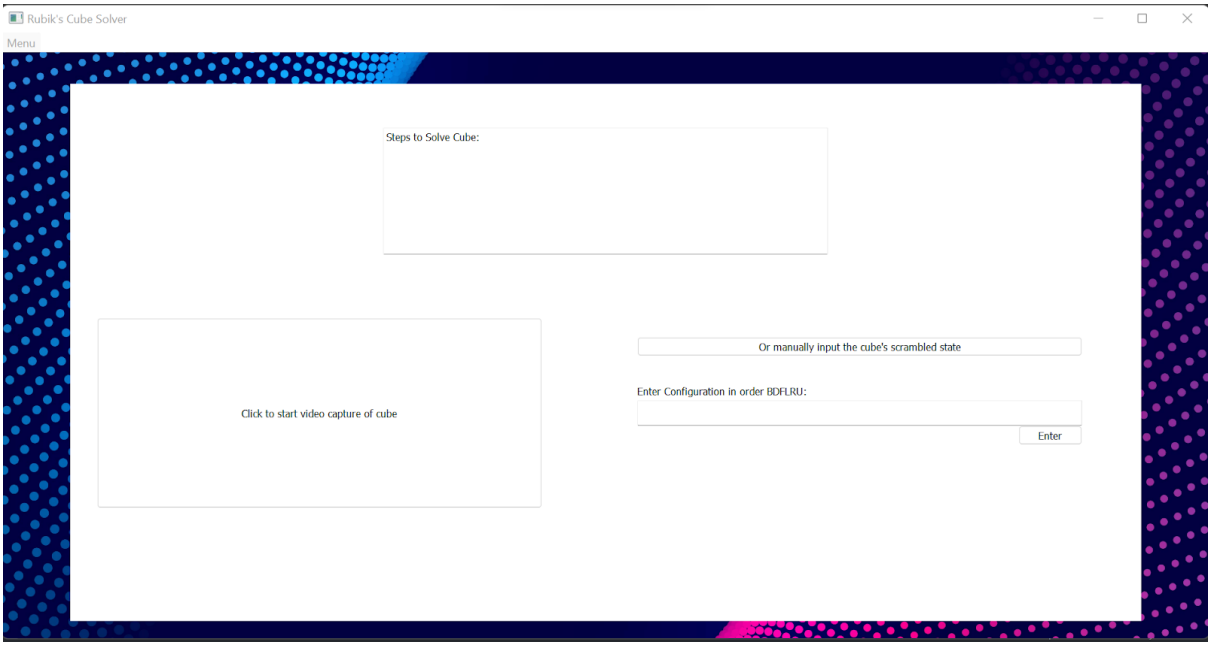
```
def test_solve_yellow_cross(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 3: ', cube.is_stage_finish_middle_layer())

    cube.solve_yellow_cross()

    print('stage: ', cube.get_cur_stage());
    print('confirm 4: ', cube.is_stage_finish_yellow_cross())

    cube.print_cube()


def test_solve_yellow_edges(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 4: ', cube.is_stage_finish_yellow_cross())

    cube.solve_yellow_edges()

    print('stage: ', cube.get_cur_stage());
    print('confirm 5: ', cube.is_stage_finish_yellow_edges())

    cube.print_cube()


def test_solve_position_yellow_corners(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 5: ', cube.is_stage_finish_yellow_edges())

    cube.solve_position_yellow_corners()

    print('stage: ', cube.get_cur_stage());
    print('confirm 6: ', cube.is_stage_finish_position_yellow_corners())

    cube.print_cube()


def test_solve_orient_yellow_corners(cube):

    print('stage: ', cube.get_cur_stage());
    print('confirm 6: ', cube.is_stage_finish_position_yellow_corners())

    cube.solve_orient_yellow_corners()

    print('stage: ', cube.get_cur_stage());
    print('confirm 7: ', cube.is_stage_finish_orient_yellow_corners())

    cube.print_cube()
```

```
def test_solve_with_known_state():

    cube = Cube(STATE_6)
    cube.print_cube()

    # cube.shuffle(10)
    # cube.print_cube()

    stage = cube.get_cur_stage()
    print(stage)

    if stage == 0:
        test_solve_white_edges(cube)
    elif stage == 1:
        test_solve_white_corners(cube)
    elif stage == 2:
        test_solve_middle_layer(cube)
    elif stage == 3:
        test_solve_yellow_cross(cube)
    elif stage == 4:
        test_solve_yellow_edges(cube)
    elif stage == 5:
        test_solve_position_yellow_corners(cube)
    elif stage == 6:
        test_solve_orient_yellow_corners(cube)
    elif stage == 7:
        print('done')


def test_solve_with_random_state():

    cube = Cube()

    cube.shuffle(50)
    cube.print_cube()

    stage = cube.get_cur_stage()
    print(stage)

    cube.solve()
    cube.print_cube()


def test_solve_with_test_state():

    cube = Cube(TEST_STATE)
    cube.print_cube()

    stage = cube.get_cur_stage()
    print('initial stage: ', stage)

    cube.solve()
    cube.print_cube()
```

```
def test_rotate():

    cube = Cube()
    cube.print_cube()

    action = input('action: ')
    while action != 'q':
        cube.rotate(action)
        cube.print_cube()
        action =input('action: ')


if __name__ == '__main__':

    test_rotate()
    test_solve_with_test_state()
    test_solve_with_test_state()
```

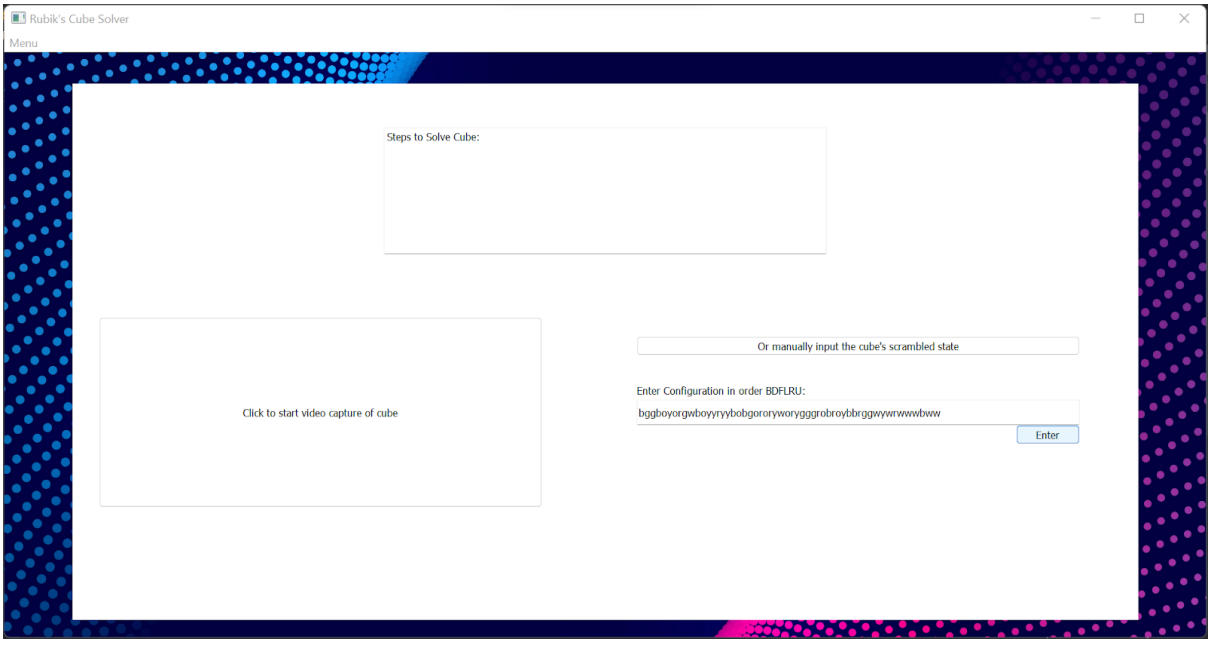
GUI

In order to display the GUI that is controlled by UserInterface.py, I have shown several screenshots below that show the process of running the GUI.

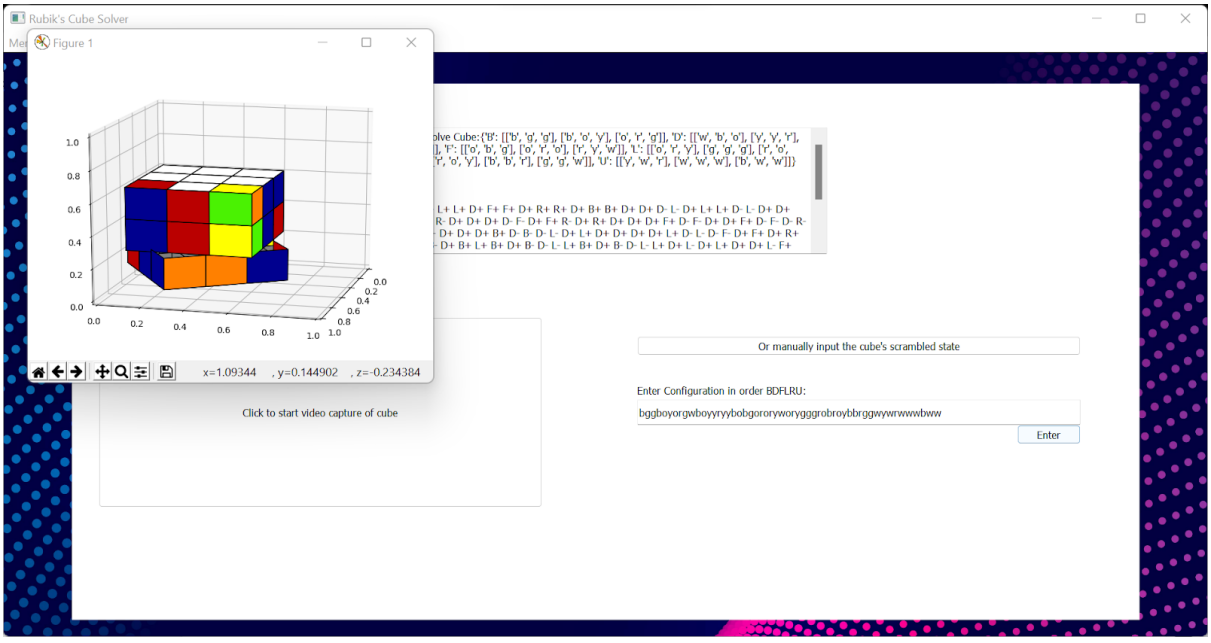


The initial configuration of the GUI when the program is first run

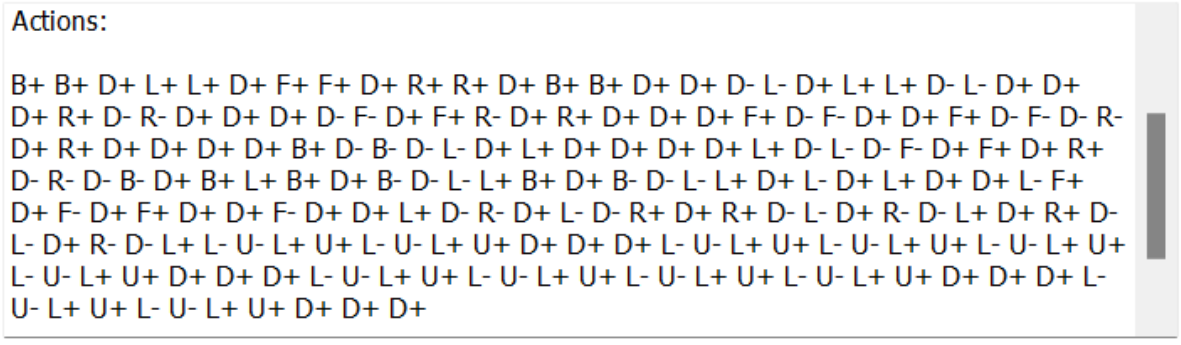
When the cursor hovers over the button, it changes colour and shape



Text is entered in the textbox for the manual state entry option



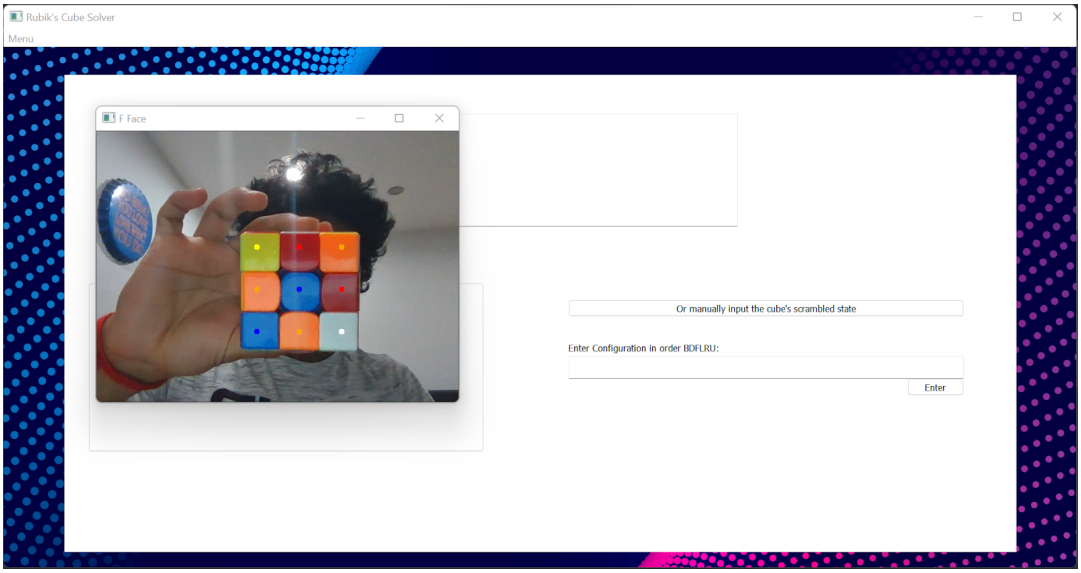
The Enter button is pressed and the textbox above is filled with the actions in cube notation and the cube model correctly modelling the rotations needed to solve it



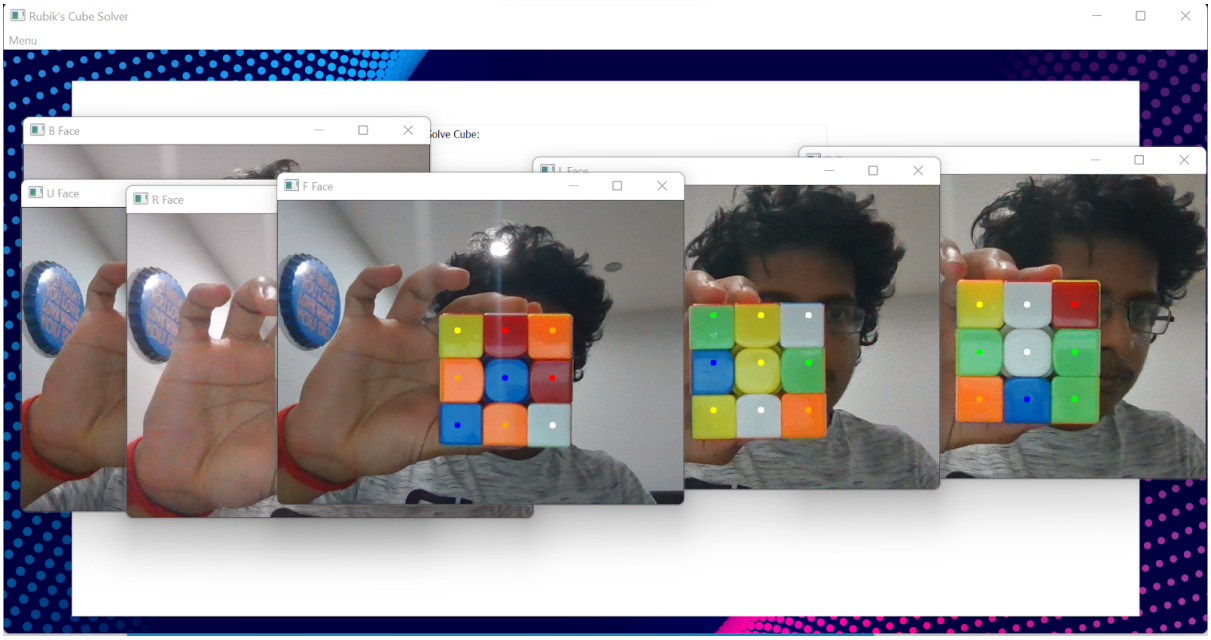
Actions:

B+ B+ D+ L+ L+ D+ F+ F+ D+ R+ R+ D+ B+ B+ D+ D+ D- L- D+ L+ L+ D- L- D+ D+
D+ R+ D- R- D+ D+ D+ D- F- D+ F+ R- D+ R+ D+ D+ D+ F+ D- F- D+ D+ F+ D- F- D- R-
D+ R+ D+ D+ D+ D+ B+ D- B- D- L- D+ L+ D+ D+ D+ D+ L+ D- L- D- F- D+ F+ D+ R+
D- R- D- B- D+ B+ L+ B+ D+ B- D- L- L+ B+ D+ B- D- L- L+ D+ L- D+ L+ D+ D+ L- F+
D+ F- D+ F+ D+ D+ F- D+ D+ L+ D- R- D+ L- D- R+ D+ R+ D- L- D+ R- D- L+ D+ R+ D-
L- D+ R- D- L+ L- U- L+ U+ L- U- L+ U+ D+ D+ D+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+
L- U- L+ U+ D+ D+ D+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+ D+ D+ D+ L-
U- L+ U+ L- U- L+ U+ D+ D+ D+

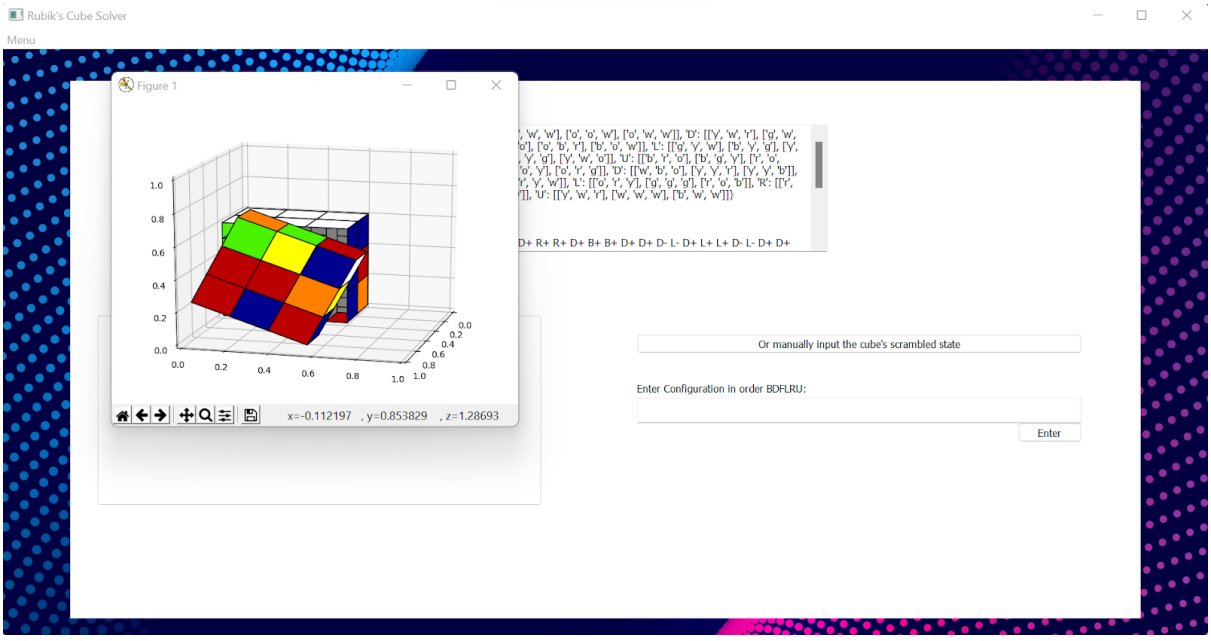
The complete actions written in cube notation in the textbox



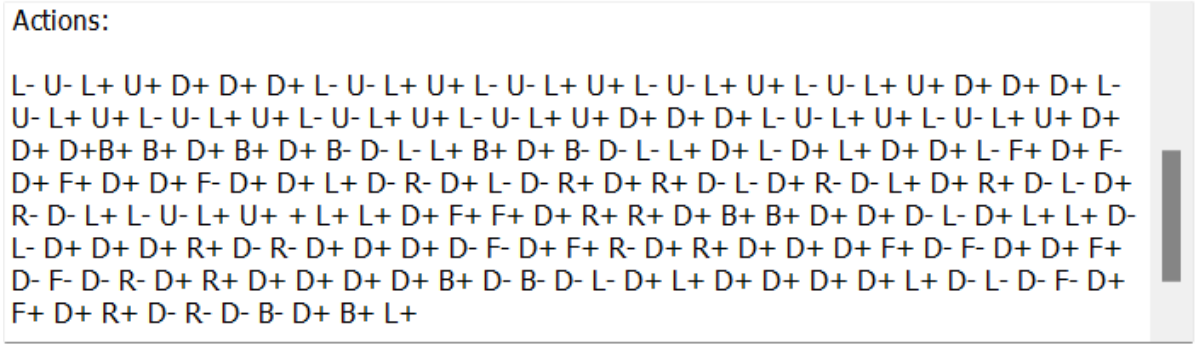
If the webcam entry is pressed, then a new window pops up, asking to show a cube face in the guidelines of the six circles. It then changes the colour of each circle depending on which colour it has detected.



This process is repeated for all six faces which have all been detected correctly



When all six faces have been correctly identified, the OpenCV windows close and the textbox above is filled with the actions in cube notation and the cube model correctly modelling the rotations needed to solve it



The complete actions would also be written in cube notation in the textbox

4 Testing

4.1 Unit Testing of Functions

Initial unit tests of modules have been carried out, much of which in parallel to development. Implementation of the GUI as well as linking functions only came after.

4.2 Requirements Testing

A sample of representative tests that demonstrate the system fulfils the requirements outlined in the specification.

Test Number	Description	Input(s)	Expected Output/ Result	Output/ Result	Matches (Y/N)
1	Enter cube face via webcam	Click enter via webcam button	New window appears to capture face	1.1	Y
2	Colour detection	Show cube face to camera	Circles on screen change to colour of each cubie	1.2	Y
3	Accepts all 6 faces	Show all 6 faces	New window appears for each face of cube	1.3	Y
4	Validates cube state	The above 3 steps	Carries on program if valid, if invalid a message is displayed	1.4	Y
5	Can enter cube state by text	Type in cube state in textbox	Accepts message in textbox and stores state	1.5	Y
6	Solving (stage 1)	Valid state inputted	Solves edge pieces of top layer of cube fully	1.6	Y
7	Solving (stage 2)	Valid state inputted	Solves top layer of cube fully	1.7	Y
8	Solving (stage 3)	Valid state inputted	Solves middle layer of the cube	1.8	Y
9	Solving (stage 4)	Valid state inputted	Solves edge pieces of bottom layer of cube	1.9	Y
10	Solving (stage 5)	Valid state inputted	Position corner pieces at bottom of cube	1.10	Y

11	Solving (stage 6)	Valid state inputted	Fully solve the cube	1.11	Y
12	Displays steps to solve	Cube state inputted via textbox or webcam	Shows solving steps in cube notation in the textbox at the top	1.12	Y
13	Displays starting state of cube as a model	Cube state inputted via textbox or webcam	Produces the correct 3D model based on the input state	1.13	Y
14	Model also displays all rotations	Cube state inputted via textbox or webcam	Produces correct rotations on the model which are detailed in the steps to solve textbox	1.14	Y
15	Exit button	Press exit button from menu	When menu button is pressed display exit button, when exit pressed, quit from the application	1.15	Y

4.3 Destruction Testing

A representative sample of destructive tests demonstrating the robustness of the system.

Test Number	Description	Input(s)	Expected Output/ Result	Output/ Result	Matches (Y/N)
16	Invalid input entered by textbox	Enter null/ invalid state in textbox	Display message in solving steps textbox	2.1	Y
17	Null input given by webcam	Enter by webcam button pressed but no faces present	Time out window with one minute passed of no faces entered	2.2	Y

18	Invalid cube state entered via webcam	Enter invalid state via webcam	Display error message in steps to solve textbox	2.3	Y
19	Illegal entering of another cube state whilst cube model is still running	Enter cube state whilst cube model not closed	Ignore the new state of the cube until the current cube model has finished	2.4	Y

5 Evaluation

5.1 Assessment of Requirements

Below is the table of specification and comments on whether each point was met. Evidence of algorithmic functionality can be found in the Testing section as well as appendix, and evidence of GUI can be found in the Technical solution section. In this project I have also added extra functionality that was not detailed in the the original spec

Specification Point	Description/subpoints	Met/Partially met/not met
Basic functionality	1) Allows user to enter cube's orientation via webcam 2) Solves cube precisely using beginner's method 3) Displays starting orientation of the cube exactly as a 3D model 4) Displays all rotations needed to solve the cube effectively via the 3D model 5) Display relevant error/warning messages	Met - 1,2,3,4 met fully. There are possibly circumstances where the most relevant error message is not displayed, but this is rare.
Cube Detection	1) The program must successfully calibrate to the colours of the client's cube 2) The program must validate the cube state that the user has inputted 3) It must identify each small square within each cube face and the colour in each square.	Most spec points are met however I encountered a few problems with the calibration of the cube's colours when the lighting of the surrounding is

	4) Must be able to run on any modern laptop with either an internal or external webcam.	changed.
Solving Algorithm	1) The algorithm must solve the cube in the format of the beginners method 2) Must state each stage of the algorithm and print which stage algorithm is currently in. 3) Program must display every move in cube notation for the user.	All spec points are met as the algorithm was successfully implemented and all moves were displayed to the user.
3D Cube Model	1) The model must accurately display the faces of the cube which have been recognised from the cube detection program. 2) The model must be able to rotate its faces by performing a rotation animation on the model 3) The model must successfully perform the correct the correct rotations for each step of the algorithm	All spec points were met as the 3D model was displayed precisely with all colours accurately represented. All rotations were also displayed successfully.

5.2 End User Evaluation

I met back with my end user (Dhruv Pattem) and asked him to use the cube solver without any instructions beforehand to see how he felt about the project and what the overall experience was like.

Generally he found the project to be extremely useful, he was able to enter in the cube colours via both the webcam and textbox successfully and he had a positive experience with it. He told me after he used it that everything went smoothly and he found the 3D model was a fantastic feature, as he could rotate his own rubik's cube to movements of the model. One improvement that he did mention was that there could be more instructions on how to input the cube’s state via the textbox. Otherwise he found the application exceedingly enjoyable and he told me that he will continue using it.

5.3 Improvements

More user-friendly GUI: As suggested, can make use of colours/icons to make the GUI more user friendly and hence appeal to a wider audience. For example perhaps the background or group box frame colours can be changed for inputting via text or webcam.

A greater range of algorithms that the cube could be solved by, will make the system a more comprehensive tool, and if they are implemented the GUI would likely have to be dynamically updated: there could be combo boxes for each group of algorithms, and upon selecting, either update a section in the GUI to display relevant details and settings for that algorithm or create a dialogue window containing this.

Later versions of this project could Include more comprehensive explanations of how each stage of the solving algorithms function and make it more clear which rotation is being done on the model.

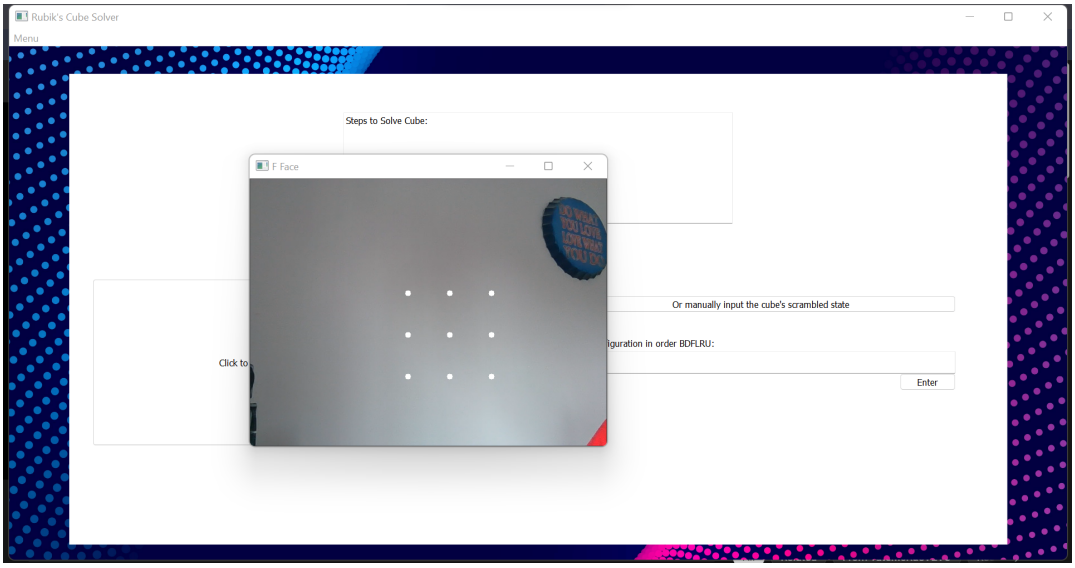
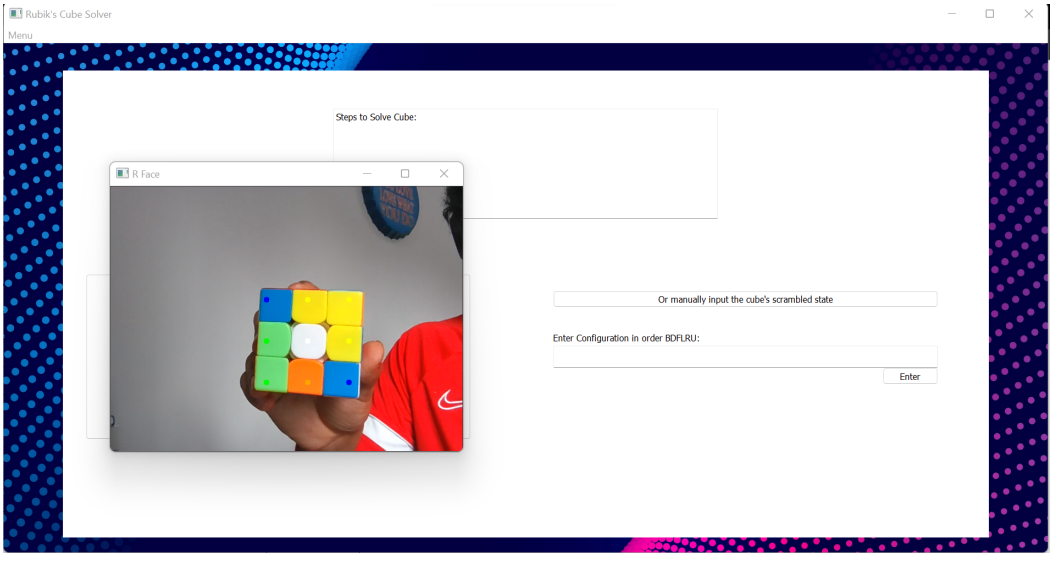
5.4 Conclusion

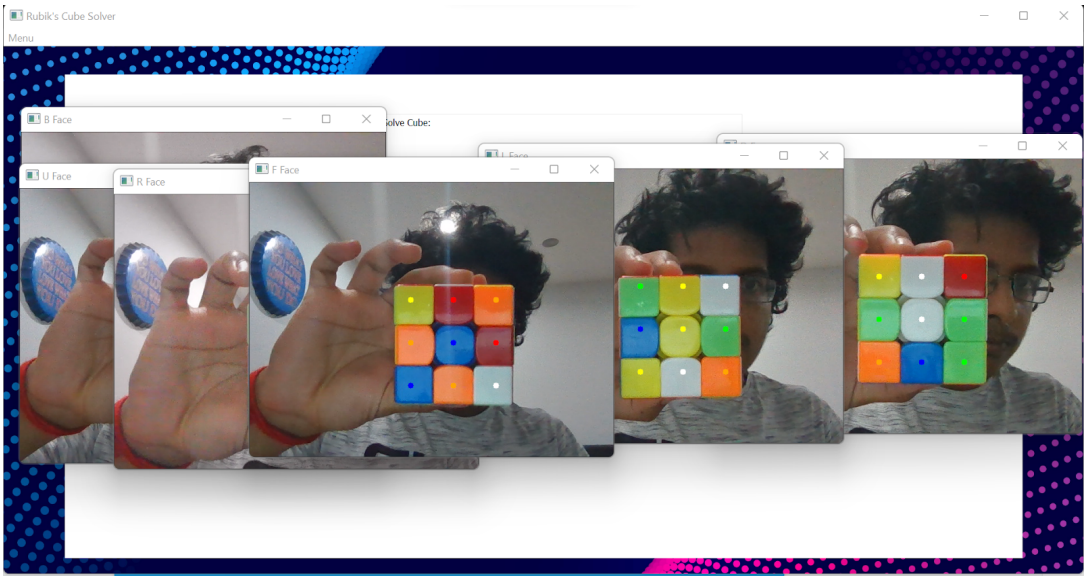
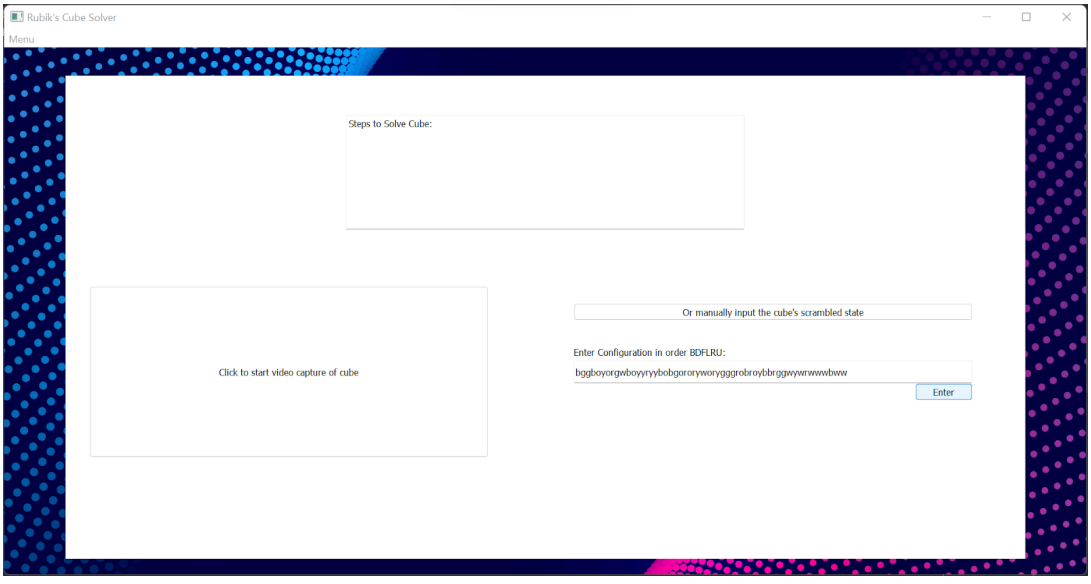
This project has been a success in completing the original specification and in its current state, can be very useful for all beginners to cube-solving. Were it developed further, the project could be extremely useful for cube-solvers of all abilities if a wide range of algorithms were used, not just the method that was used in this project.

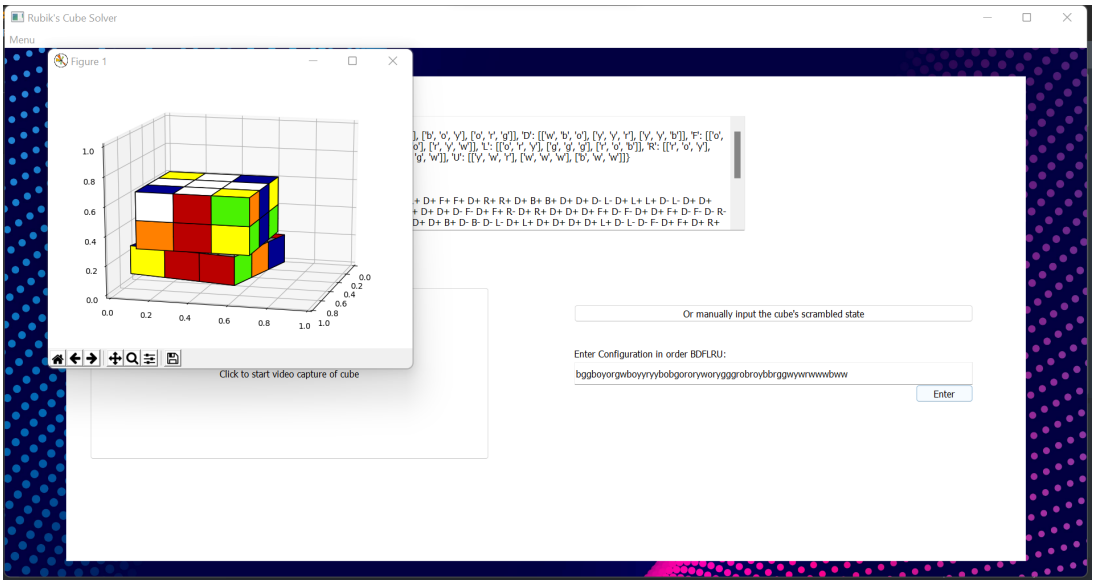
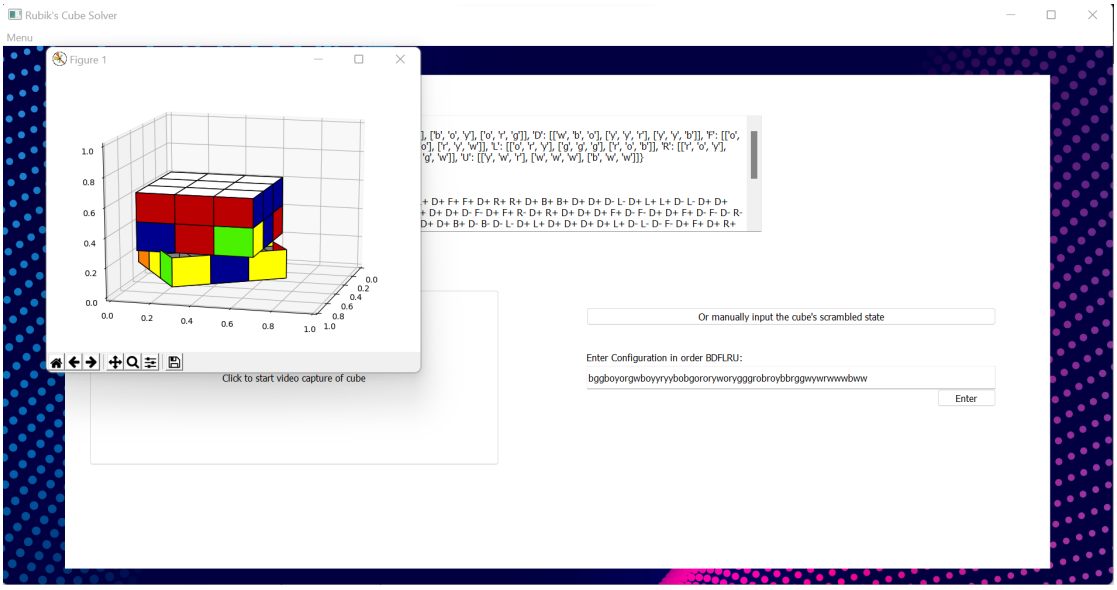
6 Appendix

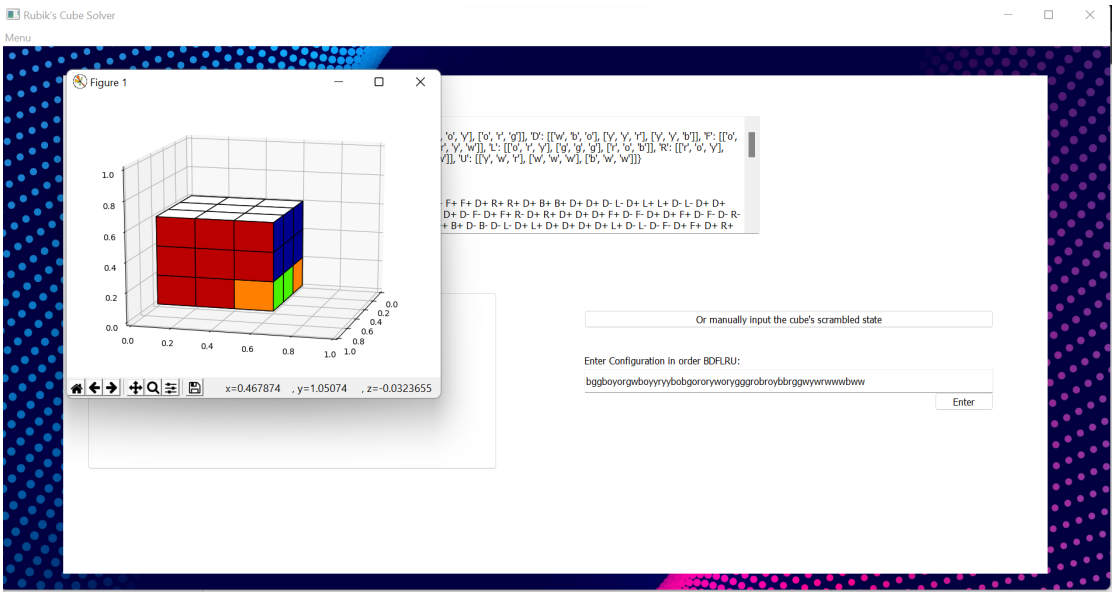
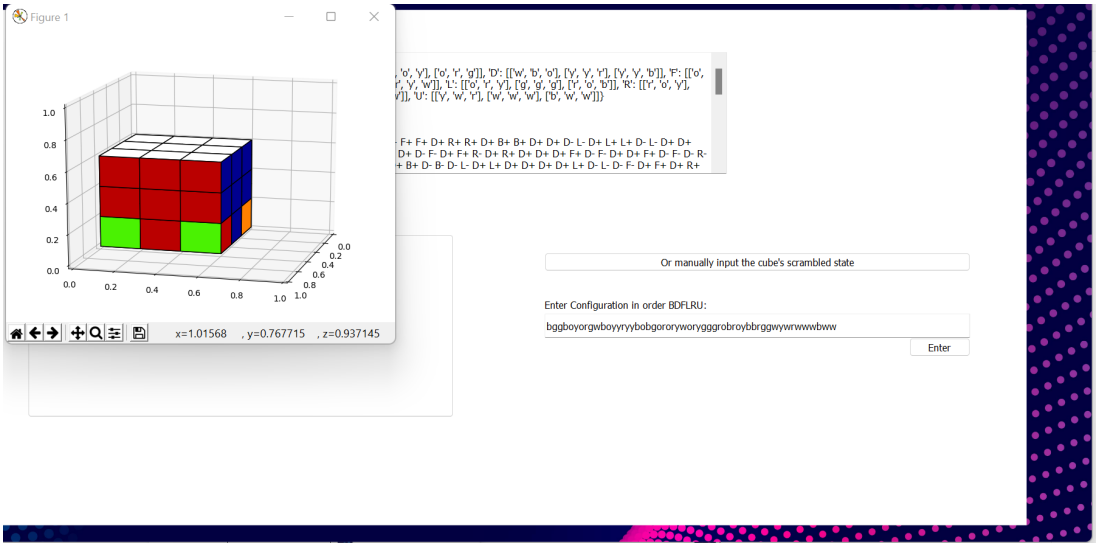
6.1 Testing Evidence

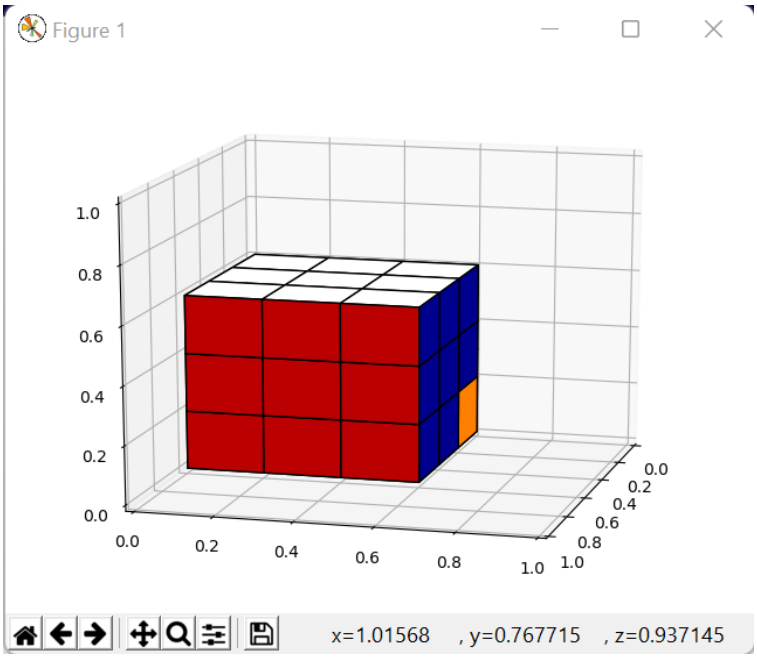
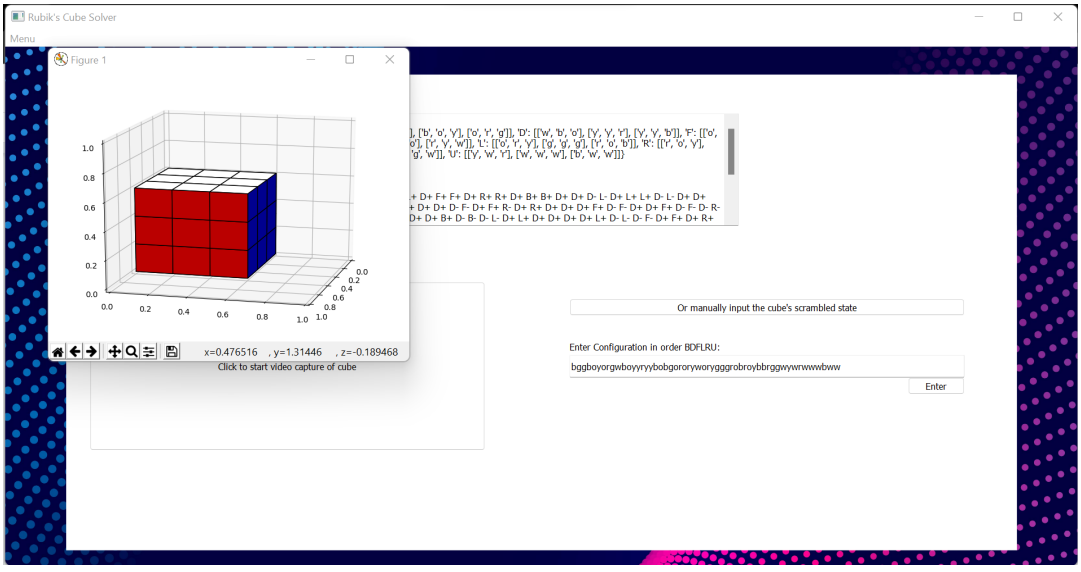
Following is the evidence that is required in the testing section

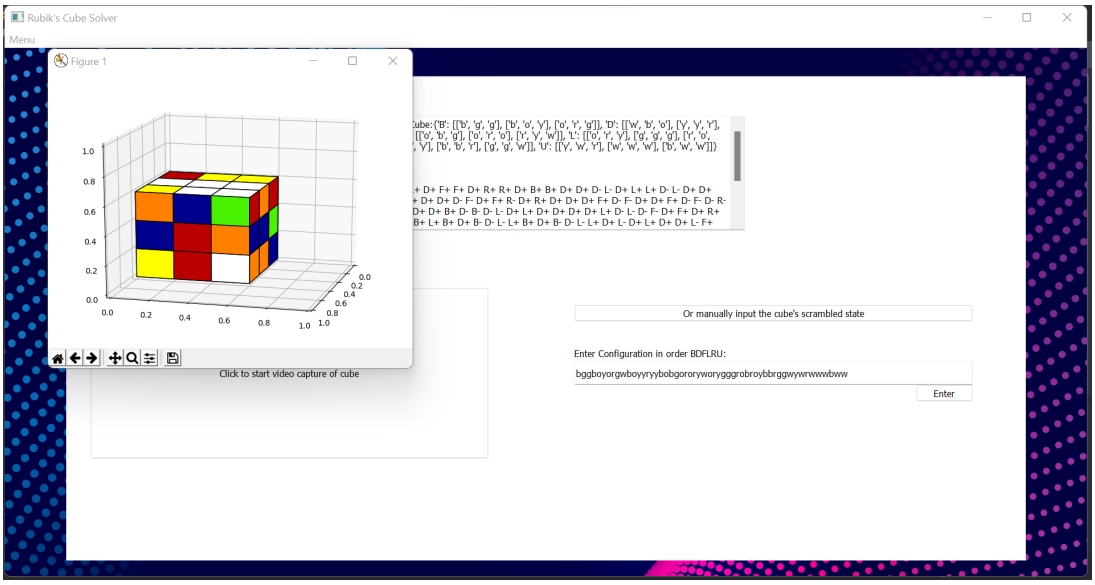
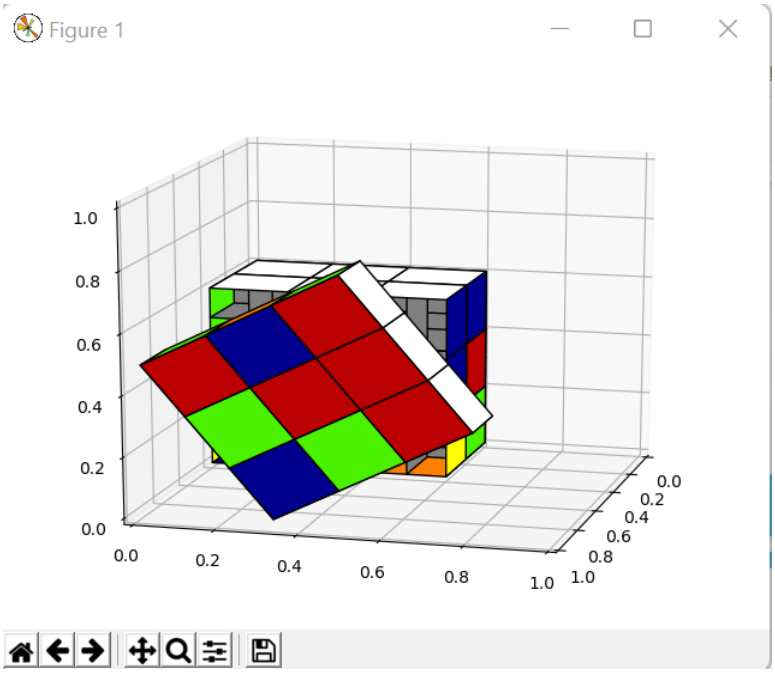
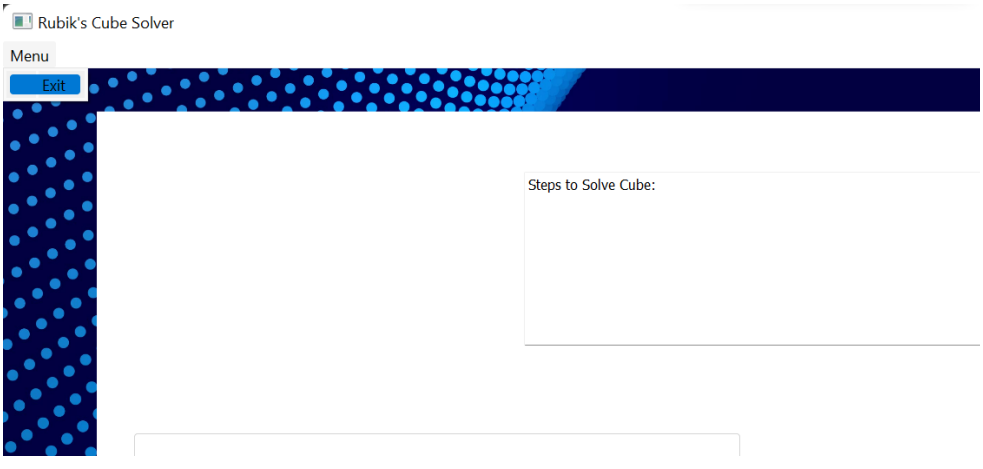
Testing Number	Testing Evidence
1.1	
1.2	

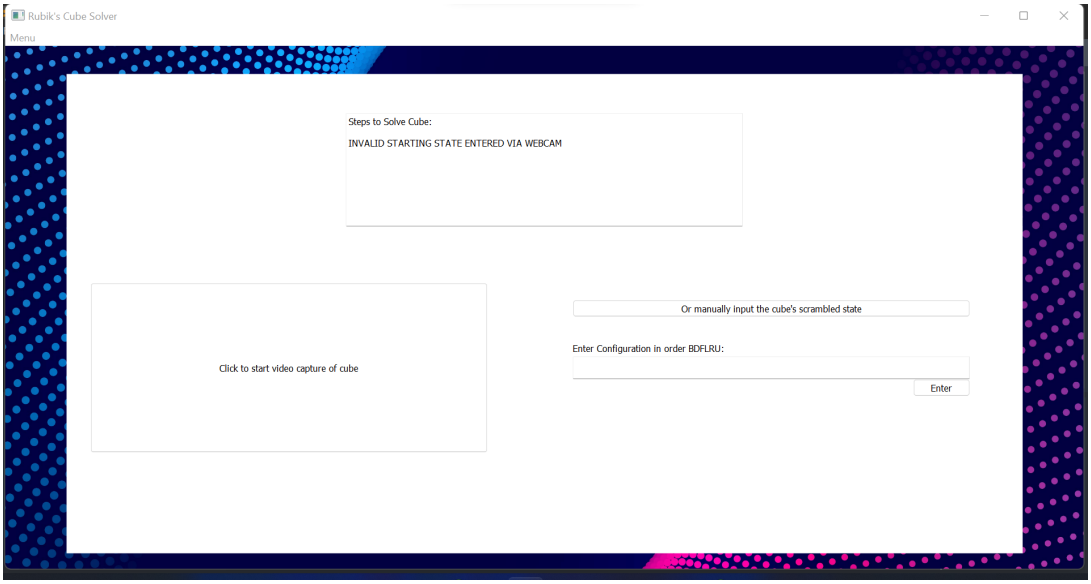
1.3	
1.4	Steps to Solve Cube:{'B': [['b', 'g', 'g'], ['b', 'o', 'y'], ['o', 'r', 'g']], 'D': [['w', 'b', 'o'], ['y', 'y', 'r'], ['y', 'y', 'b']], 'F': [['o', 'b', 'g'], ['o', 'r', 'o'], ['r', 'y', 'w']], 'L': [['o', 'r', 'y'], ['g', 'g', 'g'], ['r', 'o', 'b']], 'R': [['r', 'o', 'y'], ['b', 'b', 'r'], ['g', 'g', 'w']], 'U': [['y', 'w', 'r'], ['w', 'w', 'w'], ['b', 'w', 'w']]} Valid Starting Permutation Actions: B+ B+ D+ L+ L+ D+ F+ F+ D+ R+ R+ D+ B+ B+ D+ D+ D- L- D+ L+ L+ D- L- D+ D+ D+ R+ D- R- D+ D+ D+ D- F- D+ F+ R- D+ R+ D+ D+ D+ F+ D- F- D+ D+ F+ D- F- D- R-
1.5	

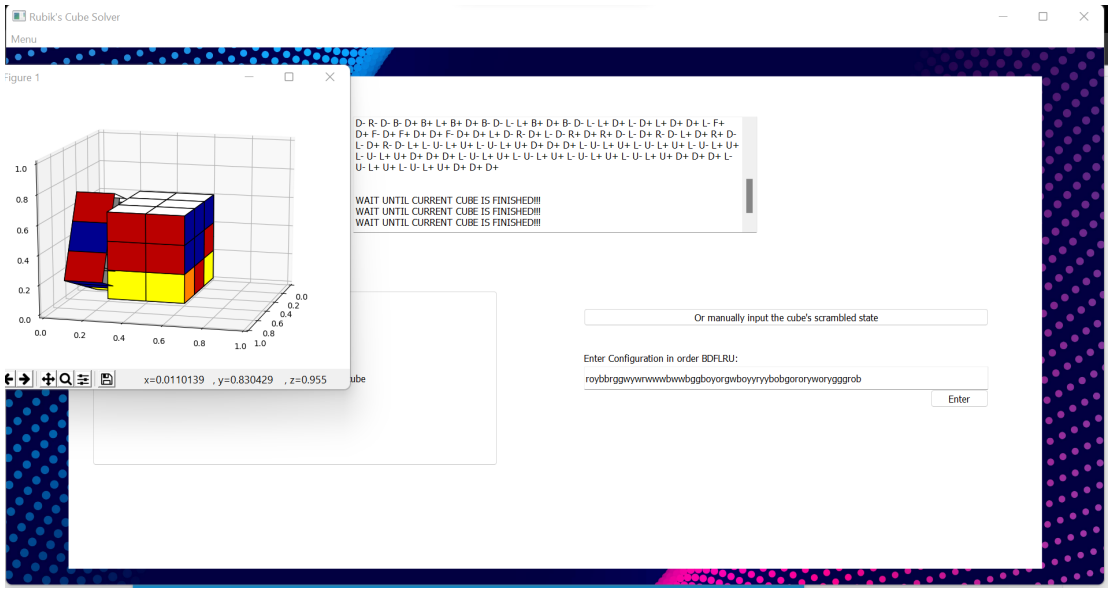
1.6	 Rubik's Cube Solver Menu Figure 1 Click to start video capture of cube Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: bggboyorgwboyyryybobgororyworygggrobroybbrggwywrwwwbww Enter
1.7	 Rubik's Cube Solver Menu Figure 1 Click to start video capture of cube Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: bggboyorgwboyyryybobgororyworygggrobroybbrggwywrwwwbww Enter

1.8	 Rubik's Cube Solver Menu Figure 1 x=0.467874 , y=1.05074 , z=-0.0323655 Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: bggboyorgwboyyryybobgororyworygggrobroybbrggwywrwwwbww Enter
1.9	 Figure 1 x=1.01568 , y=0.767715 , z=0.937145 Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: bggboyorgwboyyryybobgororyworygggrobroybbrggwywrwwwbww Enter

1.10	
1.11	
1.12	Actions: L- U- L+ U+ D+ D+ D+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+ D+ D+ D+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+ L- U- L+ U+ D+ D+ D+ L- U- L+ U+ L- U- L+ U+ D+ D+ D+B+ B+ D+ B+ D+ B- D- L- L+ B+ D+ B- D- L- L+ D+ L- D+ L+ D+ D+ L- F+ D+ F- D+ F+ D+ D+ F- D+ D+ L+ D- R- D+ L- D- R+ D+ R+ D- L- D+ R- D- L+ D+ R+ D- L- D+ R- D- L+ L- U- L+ U+ + L+ L+ D+ F+ F+ D+ R+ R+ D+ B+ B+ D+ D+ D- L- D+ L+ L+ D- L- D+ D+ D+ R+ D- R- D+ D+ D+ D- F- D+ F+ R- D+ R+ D+ D+ D+ F+ D- F- D+ D+ F+ D- F- D- R- D+ R+ D+ D+ D+ D+ B+ D- B- D- L- D+ L+ D+ D+ D+ D+ L+ D- L- D- F- D+ F+ D+ R+ D- R- D- B- D+ B+ L+

1.13	 Rubik's Cube Solver Menu Figure 1 Click to start video capture of cube Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: Enter
1.14	 Figure 1
1.15	 Rubik's Cube Solver Menu Exit Steps to Solve Cube:

2.1	 Rubik's Cube Solver Menu INVALID PERMUTATION OF CUBE ENTER AGAIN Click to start video capture of cube Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: hello Enter
2.2	 Rubik's Cube Solver Menu Steps to Solve Cube: WEBCAM TIMED OUT (>1 minute) Click to start video capture of cube Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: Enter
2.3	 Rubik's Cube Solver Menu Steps to Solve Cube: INVALID STARTING STATE ENTERED VIA WEBCAM Click to start video capture of cube Or manually input the cube's scrambled state Enter Configuration in order BDFLRU: Enter

2.4	

7 References

1. Animesh, C. Calculating the Number of Permutations of the Rubik's Cube. *Medium* https://medium.com/@chaitanyaanimesh/calculating-the-number-of-permutations-of-the-rubiks-cube-121066f5f054 (2018).
2. Aisami, R. S. Learning Styles and Visual Literacy for Learning and Performance. *Procedia - Social and Behavioural Sciences* vol. 176 538–545 (2015).
3. PyQt5 – How to add an image in Label background ? *GeeksforGeeks* https://www.geeksforgeeks.org/pyqt5-how-to-add-image-in-label-background/ (2020).
4. Qt for Python Tutorial ClickableButton - Qt Wiki. https://wiki.qt.io/Qt_for_Python_Tutorial_ClickableButton.
5. Application Main Window. https://doc.qt.io/qt-5.15/mainwindow.html.
6. OpenCV: Thresholding Operations using inRange. https://docs.opencv.org/3.4/da/d97/tutorial_threshold_inRange.html.
7. OpenCV cvtColor issue in python. *Stack Overflow* https://stackoverflow.com/questions/54737530/opencv-cvtcolor-issue-in-python.

8. Python OpenCV. *GeeksforGeeks* https://www.geeksforgeeks.org/python-opencv-cv2-cvtcolor-method/ (2019).
9. How to solve the Rubik's Cube. https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/.
10. Getting a key's value in a nested dictionary. *Code Review Stack Exchange* https://codereview.stackexchange.com/questions/201754/getting-a-keys-value-in-a-nested-dictionary.
11. mplot3d API — Matplotlib 2.0.2 documentation. https://matplotlib.org/2.0.2/mpl_toolkits/mplot3d/api.html.
12. matplotlib.collections — Matplotlib 3.5.1 documentation. https://matplotlib.org/3.5.1/api/collections_api.html.