Complete the Cube

Chiedozie Ihebuzor

Table of Contents

**Analysis1**

Problem Summary2

Background research4

Identification of end user11

Interview13

Analysis of interview14

Project objectives and use cases14

Modelling16

**Design18**

Introduction19

Class diagram21

Function listing22

Descriptions26

3D cube functions26

Mouse movements28

Cube data structures32

Solver algorithm34

Webcam41

Database47

**Technical Solution** **49**

**Testing90**

**Evaluation91**

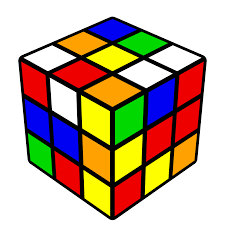
# rwari

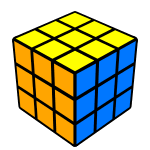
Analysis

Problem summary

The Rubik's Cube is a 3-D combination puzzle invented by Hungarian professor Erno Rubik in 1974 that gained widespread notoriety for being almost impossible to solve. The puzzle consists of a cube with 6 faces colored red, blue, white, yellow, green and orange. The faces are made of 26 smaller cubes, called cubies, attached to a center spindle. The cube can be rotated on all axis’, x, y and z. Each of the six faces can be rotated clockwise or anticlockwise. This allows for endless possibilities of movements, resulting in 43,252,003,274,489,856,000 (43 quintillion) possible permutations of the cube!

The aim of the cube is to be given a shuffled cube, meaning all the colors are in random positions, and rearrange all the cubies such that each of the 6 faces only have one color as shown in the image below.





The process of solving the cube may be even harder than you would expect. If you have ever tried to solve one before, you have promptly realized that each time you rotate one faces, 9 cubies move, not just one, this means every time you try to move a piece, it messes up the positions of your other pieces – this makes it even more of a harrowing challenge.

In order to solve the cube, you must first learn the different algorithms, patterns of rotations, that allow you to move one piece while replacing the pieces that were misplaces in the process of the translation. Typical protocol is to solve the top layer, then middle, the bottom – each row being more complex than the last as there are more positions that are messed up by each rotation.

The goal of my project is to be the ultimate tool for people learning how to solve the cube for their first times and people wanting to practice solving the cube while on their computers when they don’t have a cube with them. To solve this problem, there will be five parts to this project.

Designing a 3d model of the cube that can be controlled by inputs of the user so the user can practice solving the cube on their computers when they do not have a cube around them.

Using computer vision ai to allow the user to input the current state of their physical cube by showing the faces to the webcam.

Programming an algorithm that can solve the cube when given the current state of the cube. It will do this by following the same methods and steps a human does to solve the cube. Because the program follows the same steps the user will follow, it can give hints to the user if they request.

A robot that will use the algorithms to solve the cube for the user and scramble it too. The software will be written on an Arduino uno which will be written in the Arduino language (a language similar to C++). The Arduino will control motors and servos that will manipulate the cube.

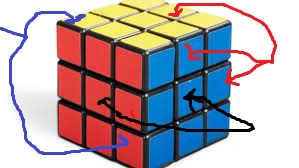
A timer that will log how quick your solves are, save the speeds and dates into a relational database which stores users and their speeds, and plot graphs with the data in the tables.

Background research

## CUBE TERMINOLOGY

# Cube mechanics

The cube is made 27 little cubes witch are called cubies, 1 centre cubie which is just for the mechanics to work, 8 corner cubies with 3 different colours on each cubie, 12 edge cubies with two colours on each and 6 centre pieces which only have one colour and can not be moves around the cube. The stickers (or coloured plates) themselves are named facelets.



Corner pieces

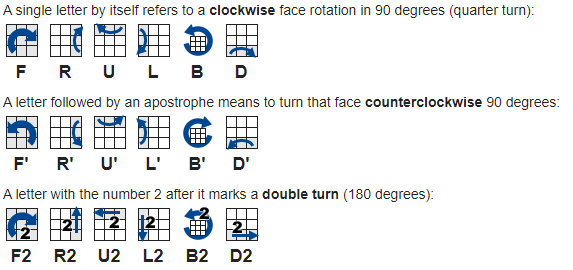
Edge pieces

Centre pieces

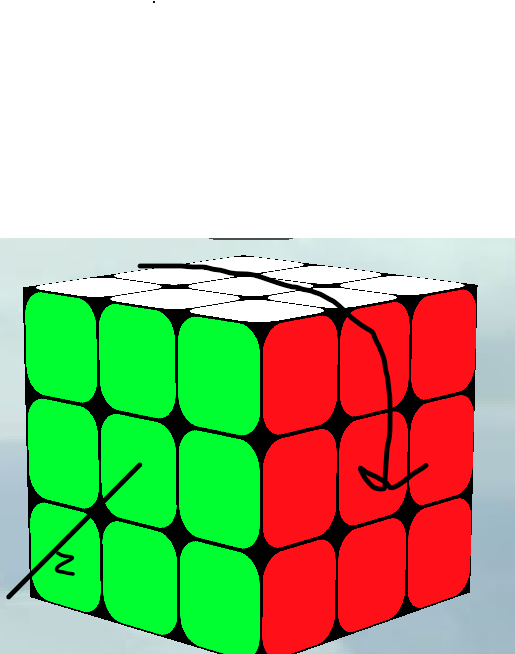
# Cube notation

In this document, I will be referring to “cube notion” often. Cubers (people who solve Rubik’s cubes) use letters to describe a rotation on the cube, to perform the solving algorithms, you must know this notation as that is what they are written in e.g. **R U R' U R U2 R' U**

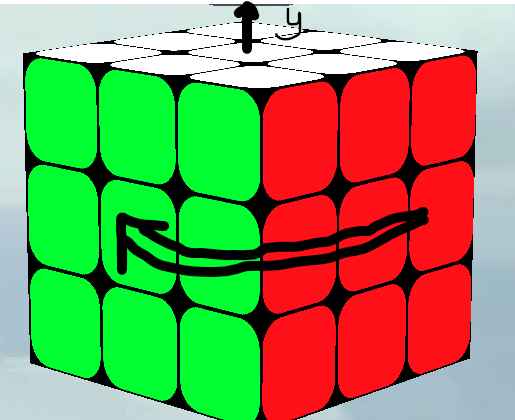
For the Beginners method, the method which the code will follow, you must know the **F** (front), **B** (back), **R** (right), **L** (left), **D** (down), **U** (up) and their inverses, e.g. R’ (pronounced as Right prime) AKA Ri (pronounced Right inverse), as explained below.



There is also notation to describe rotations of the whole cube. In this project I used the “Z” rotation, which is a rotation that rotates the entire cube clockwise about the z axis. Z2 is used to turn the cube upside down, this is important as when solving the cube, after completing the white corners, you flip the cube upside down wand work from there.



I also use the “Y” rotation and its Yi and Y2 counter parts which means to rotate the entire cube clockwise about the Y axis.



In project, I have modified this notation to describe rotations in more detail .

Rotations are done relative to their front face, performing an “F” rotation on the green face will be the same a performing a “L” rotation on the right face.

To describe which face is front, I have implemented a “Y+ colour” e.g. “YR” rotation. Which gives the axis of rotation and the colour which will be the front face after the rotation.

## EXISTING PRODUCTS – 3D SIMULATIONS

# I am the cube – by Google: <https://iamthecu.be/>

I am the cube is a 3d simulation of the cube which uses mouse inputs to rotate the faces. The mouse inputs are hard to use and complicated, this makes me think that for my product, it might be better to use a combination of mouse and keyboard to control the cube. Unlike my product, Google’s “I am the cube” doesn’t really teach you how to solve the cube, but rather just tells you facts about the cube in their “watch and learn” portion of the site, it is not a useful tool for learning.

# RubiksCu.be: <https://rubikscu.be/>

RubiksCu.be is a fantastic tool that has a lot of the same functions that mine will, it has options of both keyboard and mouse input which makes manoeuvring the cube easier than I am the cube. It also has a solve section of the page which allows the user to input the current state of their cube by clicking on a colour and then the square to assign that colour to, then the website solves the cube. The method of inputting the current state of the cube was tedious and painstaking, in comparison, my method of using an ai in the webcam will be much more convenient for the user and a much faster way of inputting the data. Another point is that the algorithm RubiksCu.be uses – the Kociemba Solver Algorithm - is not “human friendly”, it is an algorithm that is efficient for computers but is very difficult for a human to pull off in their head, this means that by showing the directions to solve the cube with that algorithm, the user isn’t learning anything and won’t make any progress, whereas with my product, the computer will use the same algorithms that a beginner would use so that as the user is following the computer generated solution, they are able to understand what is going on

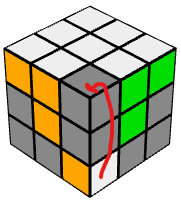
## Solving Algorithms/Methods

The are many ways to solve the cube. In this portion of my analysis, I will review the top 3 solving methods and explaining the pros and cons of each.

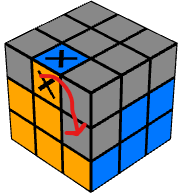
# 1 The Beginners method

The Beginner’s Method (AKA layer by layer) is the first method people learn, hence the name, it involves solving top layer, then the middle layer and lastly the bottom, hence the name. It is suitable for beginners as is it is intuitive and simple to understand.

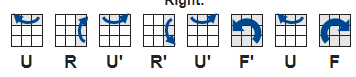
1. White cross - Create a white cross pattern on the top by lining the white edge piece with the centre pieces.



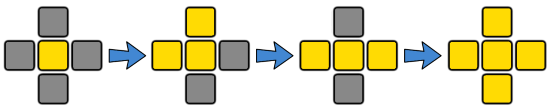
1. White corners – use the Ri, Di, R formula to move the white corners into a place where both each of the sides matches the centre colour of the face it is on.



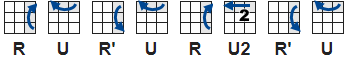
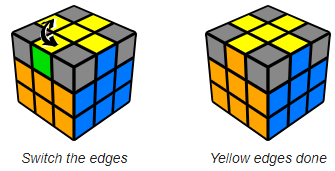
1. Middle layer – uses the U, R, Ui, Ri, Ui, Fi, U, F formula to move an edge piece to the right and its inverse Ui, Li, U, L, U, F, Ui, F to move the edge pieces to the left, which solves the solve middle layer.



1. Yellow cross – once the first two layers are solved, the next step is to solve create a yellow cross pattern. You can get three possible patterns on the top. Use the algorithm below to go from one state to another, the second diagram shows how performing the algorithm on each state progresses it to the next and eventually to the yellow cross

1. Yellow edges - After making the yellow cross on the top of the cube you have to put the yellow edge pieces on their final places to match the colors of the side center pieces. Switch the front and left yellow edges with the following algorithm

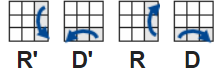
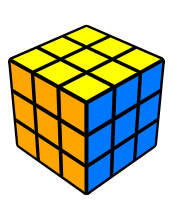
1. Position yellow corners – to solve the last layer corners, first we have to get them to the right spot, orientation doesn’t matter in this step. Find a piece which is already in the right place (if there are non in place, perform the algorithm on any side), move it to the right-front-top corner then apply the following algorithm to switch (cycle) the three wrong pieces marked on the image.

This algorithm switches 3 corners as shown in the image





1. Orient yellow corners - all the pieces are in the right positions, all that remains is to orient the yellow corners to finish the cub. Hold the cube in your hand with an unsolved corner on the front-right-top corner then perform the algorithm below twice, or 4 times until the piece is oriented so that the yellow side is on the top. Rotate the top (U) face to get another unsolved piece into the front-right-top corner and repeat. Do this will all unsolved corners and the cube is complete.



# 2 cfop

The CFOP method is the method used by speed-cubers (professionals who practice solving the cube in as little time as possible). The acronym stands for Cross, F2L (First Two Layers), OLL (Orient Last Layer), and PLL (Permute Last Layer).

1. The Cross

The first part of this algorithm is to solve the cross on the first layer, just like the RCSG method but here the cross does not have to be white. On average this takes about 7 moves to accomplish

1. F2L

The next part is to orient the corner pieces on the first layer correctly while simultaneously orienting the edge pieces in the middle layer to correspond with the corner pieces. This also about 7 moves on average, for each of the corners.

1. Orienting of Last Layer

The second to last part is to orient the corner and edge pieces of the last layer simultaneous so that the last layer has the right color. There are 40 different algorithms for this part and takes about 9 moves to complete.

1. Permutation of Last Layer

The last part is to, without disturbing any of the corner or edge pieces, permute the remaining 8 pieces at once. There are 13 different algorithms for this part, and it takes about 12 moves to complete.

# 3 Kociembas algorithm

The Kociemba Algorithm is the mathematically most efficient, quickest method of solving The Rubik’s cube. It solves the cube in 20 moves on average, which takes an average of 110 moves to solve. The method was created by Herbert Kociemba and was optimized by math to be made for computers to use. It is not practical for a human to use this method as it is not intuitive in the slightest. Its best use case is for robots solving the cube, such as the “Sub1Reloadded” robot which solved the cube in only 0.637 seconds in 2016, taking the world record for the fastest robot to solve the Rubik’s cube.

## EXISTING ROBOT DESIGNS

# 6-axis robot design

The 6-axis design consists of 6 stepper motors which both can rotate 360 degrees in both directions. We have one stepper motor for each side of the cube meaning there is no need to rotate the entire cube. This design is the easiest as the programming is very simple, we can translate cube notation directly into stepper motor movements. For example, D’ would be the bottom stepper motor rotating 90 degrees counter-clockwise. This method would also be the fastest because there is never a need to reposition the cube, each rotation of the motors means a move, hence why the world record holding robots use this design. A con if this design is that it very expensive, good stepper motors are costly and having 6 of them would make the price very unreasonable for the consumer.

# Arduino Robot Rubik cube Solver - YouTubedual servo design

The servo deign consists of 2 servo motors (small precise motors that can only rotate 180 degrees) The base servo rotates the cube then the arm servo holds the cube in place while the base servo rotates the bottom face. The arm servo pushes the cube so that it topples over, and another face can become the bottom face, meaning that face can be rotated. This design must use complex software and algorithms to figure out how to position the cube in the correct orientation by a combination of twists and pushes from both servos. Due to the amount of movement of the motors needed to make one face rotation on the cube, this method is very slow. This is not helped by the fact that servo motors are significantly slower than stepper motors, however, servo motors are much cheaper and only two are needed in this design.

To provide the best user experience, I have chosen the dual servo design for the project. A deciding factor was that the user would receive more satisfaction from seeing the cube being solved with this quirkier method and it would more entertaining to watch as there is more motion.

## GODS NUMBER SCRAMBLE

The term “Gods Number” refers to the maximum amount of turns it takes to solve a Rubik’s cube. This number was proven using 35 CPU-years of computer time donated by google. With this knowledge of gods number, I can design a scrambling algorithm that will produce a set of 20 moves for either the robot or the human to perform.

Identification of end user

The Rubik’s Cube’s recommended age group is around 3+ in fact, the world record for the youngest person to solve the cube is held by Ruxin Liu since April 2013 when she solved the cube in under 2 minutes at the age of 3, however, most three-year-olds are not able to solve the cube as it involves memorisation and spatial awareness skills. The reason the age ration is so low is simply that it does not provide much or a chocking habit and therefore can be used by younger children.

Hasbro, a popular toy company, recommends the cube to children ages 8 and up. This rating better accounts for the mental capacity needed to learn to solve the cube, therefore I will be referring to 8+ as users age. With these ages in mind, the user interface will have to be altered so that it is simple enough to be operated by a child, to do this, the hints given will be both written and visual so that children will be able to understand them. I will also use a lot of bright colours in the colour scheme of the program to engage the younger audience. Another way the product caters towards the younger users is the methods of data input. The mouse controls are intuitive they are design to mimic the way you would manipulate the physical Rubik’s Cube. The current cube stare input is also intuitive as all the user must do is point the cube at the webcam and rotate to show each of the faces.

Just because the project is made to be suitable for younger children does not mean that the product is made for younger children. All ages can use this product and benefit from it equally

The Rubik’s cube is not targeted at any specific gender; however, most “cubers” (people who solve the cube) are male. My product will be focused on both genders equally. To accommodate female users, I will include an option to change the cubes colour into a more feminine cube with pastel colours. These options will be in the main menu and will change the entire look of the UI (user interface) too.

Feminine cube design

What type of people is the product aimed at?

* People who have never solved the Rubik’s Cube before and are looking to learn in an intuitive modern way.
* People of all skill levels who want to practice solving the cube on their computers when they do not have a Rubik’s Cube on them.
* Beginners looking to sharpen their skills and memorise the methods and algorithms.
* People who just want to see a cool robot and play with it for fun.

How will my project accommodate these people?

* Robot will solve the cube when the user gets stuck to show how to complete it, this is useful for beginners.
* Program will give well written visual hints depending on what stage of completion the user is currently at.

Interview

To get a better idea of the needs of the user, interviewed a friend from college, age 17, who can solve the Rubik’s cube.

What degree of customisation of any would you like to see over the cosmetics of the model?

* Changing the cube colours would be pretty cool but it isn’t really necessary, changing the size of the model so you could look around more easily would be helpful.

What controls would you like?

* Click and drag, I have no clue how you’d even start coding that, maybe “w, a, s, d, q and e” keys, for the 3 axes.

Are there any features you would like?

* Saving cube layouts so you could watch the model solve a certain pattern multiple times, that might be useful if you keep getting stuck on the same bit.
* Choosing the solving method you want, like CFOP etc. it wouldn’t need to be implemented but it would be a nice option to come up for the hints.

How would you use the robot?

* It could show how a set of moves works and could help you understand it for yourself, especially if you can slow it down.

Which robot design intrigues you the most?

* The 6 axis design looks more interesting to me however I wouldn’t want to pay the price for it so I would choose the dual servo design

What informative method will teach you how to use the cube?

* I learnt how to solve the cube with videos.

ANALYSIS OF INTERVIEW

From this interview, we can establish a list of several suggestions that can be utilised to help me solve the problem. The first new suggestion was saving cube states, this would allow to user to repeat the same section of the cube repeatedly for practice. To do this, the program would have to contain a text file in which the cube state is held in a json format. The second feature put forwards was the ability to control the speed at which the robot solves the cube, the idea being that the user could analyse the moves better at slower speeds, to achieve this, I will connect a potentiometer dial to the Arduino controller and the turning the dial up will reduce the delay between each move of the (servo) motors. The interviewee mentioned that they initially learnt how to solve the cube by watching a YouTube video, the advantage being that instead of looking at a static image, they could watch the turns they needed to solve the cube. To take advantage of this, I will create animations for the hints to make them easier to understand as opposed to a static image. Lastly, the interviewee suggested that they would like to be able to choose which methods the hints they receive are following. I think this is a great idea so I will be creating hints not only for the beginner’s method, but for the CFOP method too.

Project objectives.

## THE CUBE

* Design a data structure that holds the current state of the cube.
* Make a function that figures out the new state of the cube after each rotation and updates the data structure.
* Create a 3d model that displays a simulation of the cube in 3D has animations for rotations.
* Allow user to drag mouse around screen to make the cube rotate around its axis’.
* Map the path of the mouse movements when the user drags the mouse on the cube to figure out which face they are trying to rotate.
* Allow user to use keyboard to rotate faces (D = Right, A = Left, W = Up, S = Down, Q = Front, E = Back + and any of these letters with shift for the anticlockwise rotation of the face).
* Allow users to undo a move by storing previous moves in a stack data structure and popping from the stack to undo a move.

## THE SOLVER SCRIPT

* Write code finds the solution to the Rubik’s cube when given the variable of the state of the cube. The code will solve the using the Beginners Method, the same one a human would use, and return cube notation for the solve.
* Make the solver script able to give hints of what set of moves the user must use next based on it analysing what stage of completion the cube is on (e.g. if the white cross is formed, the program will see than and respond by giving the hint “move the white corners to complete the white face using the U’ R’ U R, or U L U’ L’ algorithms).

## THE HINTS

* Create good visual hints.
* Design animations to show the next steps needed.

## COMPUTER VISION

* Write code to allow webcam to locate the cube.
* Design program to let webcam identify and store the colours on the cube and their positions into a variable.
* Prepare prompts and instructions telling the user to rotate the cube to show all 6 faces in the right orientation.

## The Robot

* Construct robot frame and wire the electronic components.
* Translate cube notation into a set of moves for the robot to execute.
* Write 20 move random scramble algorithm for robot to follow.
* Create potentiometer inputs to slow down the speed of the robot.

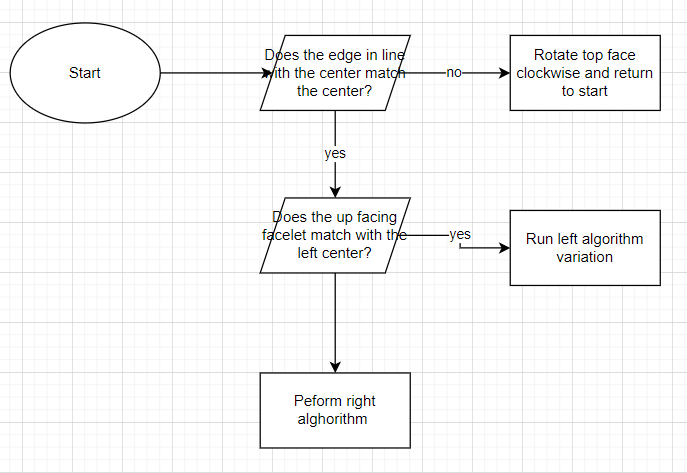
## THE TIMER

* Design a relational database to save the dates and speeds of solves of each user.
* Create graphical outputs of speed to show the user their progression.

Modelling

# This python code creates an xyz position for each cubie, 0,0,0 being the centre piece

cubies = []  
for x in range (-1,2):  
 for y in range(-1,2):  
 for z in range(-1,2):  
 cubies.append((x,y,z))

This flowchart shows the kind of logic that will be used throughout the program to solve the cube. This flow chart in particular shows the process of solving the second layer.

Design

Introduction

In this section of the documentation, I am describing the design of Complete the Cube.

# Programming language

My programming language of choice is python, I chose python because it is a very versatile language, featuring an infinite range of libraries for any need

# Libraries

I am using the Ursina library for the 3D render of the Rubik’s Cube because it is a simple library and has a fresh clean look to it.

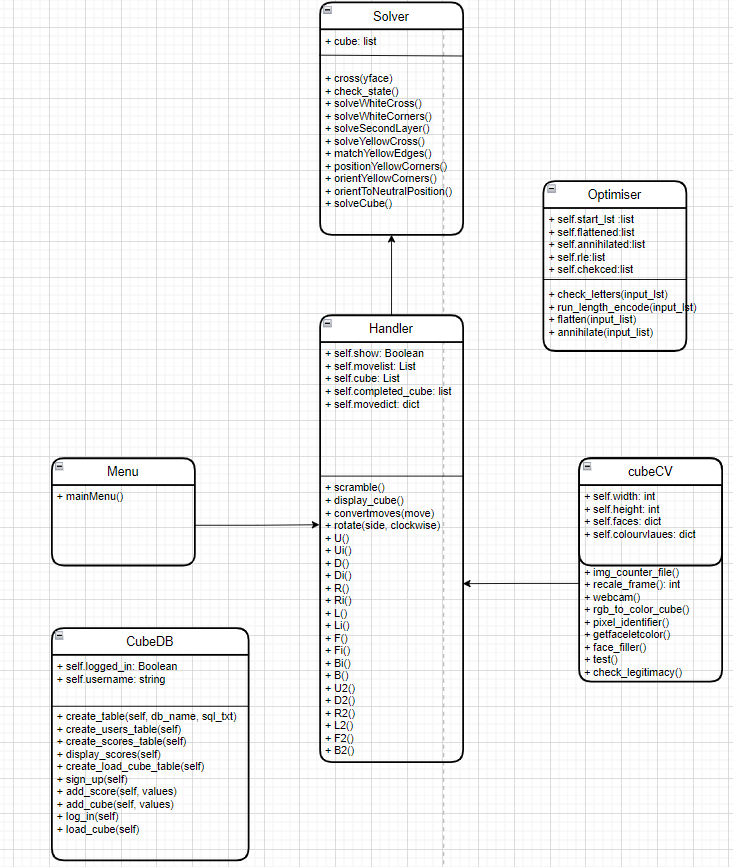
For the computer vision part of the code, I used OpenCV

# Hierarchy chart

A picture containing text, white

Description automatically generated

# Class Diagram



# Function Listing

Cube3D class:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| Initcube | None | None | Creates 27 cubie entities and positions them into a 3 by 3 by 3 cube, appending them into a list |
| Reset | None | none | Destroys all cube entities and empties list of entites |
| Save | None | None | Saves the rotation of each cubie entity into a list |
| Load | saved | None | Creates 27 cubie entities with rotations taken from a save list |
| runlist | Sequence:list | None | runs through sequence of moves in a list |
| rotator | key | None | Rotates a face of the cube |
| input | key | None | Ursina function takes in key presses and is always running |
| parent\_child  \_relation | Axis, layer | None | Attaches 9 cubies on a side onto the center of the scene |
|  |  |  |  |
| gradient | Start, end | None | Finds out what move to do based on a drag movement |
| Get front face | None | Front\_face  :string |  |
| adjustment | Front\_face:  string | Adjustment  dict: dictionary | returns dict with what moves to do when the front face changing |
| Upside\_down | Move:str | Adjusted\_move | Returns the equivalent move to a move done on an upside down cube |

Handler class:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| Update\_cubies | None | None | Updates cubie vairables |
| scramble | None | None | Perfomers 20 randommooves on the cube to scramble it |
| printCube | None | None | Displays cubes nicely |
| Display\_cube | None | None | Displayes cube nicely |
| runthrough | seqence | None | Perfrorms sequence on moves |
| Cube\_to\_rotations | None | rotations | Finds put the rotation of each cubie in a list |
| convertmoves | move | None | Given a string, performs calls the rotation function |
| U,R,L,F,D,B0 | None | None | Performs clockwise rotation on the faclet model of the cube |
| Ui,Fi etc | None | None | Performs anticlockwise rotations |
| Y/Yi | None | None | Rotates entire cube about Y axis clockwise |
| Z2 | None | None | Rotates entire cube upside down |

Solver class:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| cross | Yface:string | Cross:Bool matched:Bool | Determines if a face has a cross and is matched with the layer below’s center |
| Check\_state | None | State: list | Returns which stage of solved the cube is on and the next stage |
| solveWhiteCross | None | None | Solves white cross |
| SolveWhiteCorners | None | None | Solves the white corners |
| solveSecondLayer | None | None | Solves the second layer |
| solveYellowCross | None | None | Solves the yellow cross |
| matchYellowEdges | None | None | Matches the yellow edges to their centers |
| Srt (lambda) | Piece:  dictionary | Sorted pieces:  list | Sorts pieces on cubie into list |
| checkdone | None | boolean | Checks if yellow corners arein the right position |
| orientYellowCorners | None | None | Orients yellow corners |
| OrientToNeutral position | None | None | Orients cube such that white is on top, and green is on the bottom |
| solveCube | None | None | Goes through all steps and solves the cube |

Optimiser class:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| check\_letters | Input\_lst: list | Output\_lst:list | Changes R3 to Ri |
| run\_length\_encode | Input\_lst: list | Output\_lst:list | Run length encodes input\_lst |
| flatten | Input\_lst: list | output: list | Expands R2 to R,R |
| annihilate | Input\_lst: list | Input\_lst: list | Cancels R,Ri into nothing |

CubeCV class:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| Img\_counter\_file | None | Count: integer | Reads counter file and returns the next number to use for image names |
| Rescale\_frame | Percent: Integer | Resized frame | Resizes webcam box size |
| webcam | None | None | Opens webcam and asks users to take picture of cube |
| Rgb to color cube | None | None | Turns a cube 3d list of rgb values into a cube of colors |
| Pixel identifier | Coordinate: tuple, file: string | Rgbpixel: tuple | Returns color of the pixel in the coordinate position |
| getfaceletcolor | Coordinate: tuple  file: str | Avgcol: tuple | Returns average rgb value of a facelet |
| Face\_filler | Face: list  Color: tuple  File: str | None | Inputes rgb valies of facelet into the 3D list |
| Check\_legitimacy | None | boolean | Checks if inputed cube is possible |

Menu class:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| mainMenu | None | None | Prints menu and lets user choose what to do |

CubeDB class: (In the rubiks\_database.py file)

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return | Description |
| Create\_table | db\_name: str  sql\_txt: str | None | Creates a table in the database |
| create\_users\_table |  |  | Creates table to store user data |
| create\_scores\_table |  |  | Creates table to store scores |
| create\_load\_cube\_table |  |  | Creates table to store saved cubes |
| display\_scores |  |  | Displays the scoreboard |
| sign\_up |  |  | Adds users information into the database |
| add\_score | Values:list |  | Adds a score to the database |
| add\_cube | Values: list |  | Adds a saved cube to the database |
| load\_cube | Cubie\_rotation:  List  Move\_history:  list |  | Allows the user to choose a saved cube to load |
| log\_in |  | Boolean | Logs the user in if there is an account and directs user to sign up if not |

Descriptions

# 3D Cube

Making the cube

The 3D cube consists of 27 Cubies placed to form a 3 by 3 cube. The cubies are 1 by 1 in size and are positioned on a 3D graph with origin point(0, 0, 0). Each cubie is an ”Entity” with attributes of a 3D blender cube model, a **texture png of a cube (figure 1)** net and size of 0.5. when the 27 cubies are placed together, they form a completed Rubik’s cube.

## A picture containing bubble chart Description automatically generated

Figure

DEFINE InitCube():

FOR i in range(-1,0,1):

FOR j in range(-1,0,1)

FOR k in range(-1,0,1)

Entity(position=(i,j,k), scale=0.5)

Rotate side

To rotate a face, the cubies on one face are attached to the center so that when the center cubie is rotated, the attached cubies rotate along with it in a coherent manner. The parent\_child\_relation() function sets the parent of all cubies on one side of one axis to the center; then the rotator() function rotates the center.

DEFINE parent\_child\_relation(axis, layer):

FOR cubie IN cube:

IF cubie.position.{axis} == layer

cubie.parent = center

To rotate the face, for each rotation move, we need to know which axis the face will rotate on, which side (negative or positive) if the axis does the face lie on, and by what angle to rotate the cube. This information is stored in the rot\_dict dictionary the form {‘move’:[axis, position, rotation]}

rot\_dict = {'u': ['y', 1, 90], 'd': ['y', -1, -90], 'l': ['x', -1, -90],

'r': ['x', 1, 90], 'f': ['z', -1, 90], 'b': ['z', 1, -90]}

DEFINE rotator(key):

axis, layer, rotation = rot\_dict[key]

parent\_child\_relation(axis, layer)

if held\_keys["shift"]:

rotation = -rotation

center.animate\_rotation\_{axis}({rotation}, duration = 0,4)

Reset

The user must be able to reset the cube to start over. To reset the cube, all entities are destroyed and then replaced in the manner shown below.

FOR cubie IN cube:

Destroy cubie

Cube.Clear()

InnitCube()

Save

One of the parameters of the cubie Entity is its rotation. A cube can be rotated in 24 possible ways. The rotation of the cubie can be modelled as a tuple with (x rotation, y rotation, z rotation), for example, (0,-90,180) where -90° is 270°.

The cube is saved by getting each of the rotations of the 27 cubies then saving them into a list, saving the cubes in reverse order.

DEFINE save():

global saved

rotations = []

for i in reversed(range(27)):

d = cube[i]

i, j, k = d.rotation.x, d.rotation.y, d.rotation.z

rotations.append((int(i), int(j), int(k)))

saved = rotations

Load

To load a saved cube, the cube is cleared. Then each cubie entity is created with a xyz position and a rotation which is gotten from popping the last favlue from the saved list. As the saved rotations are reversed, performing .pop() on the saved list will load the cube in the correct order.

for x in range(-1, 2):  
 for y in range(-1, 2):  
 for z in range(-1, 2):  
 pos = (x, y, z)  
 inst3D.cube.append(  
 Entity(model=**'cube\_model.obj'**,  
 texture=**'cube\_texture.png'**,  
 position=pos,  
 rotation=cubie\_rotations.pop(),  
 scale=0.5))

# Mouse movements

To make the cube simulation feel natural to manipulate, I have adopted a click and drag method, right-click and drag to move the camera; left-click to rotate a face. The right-click function in built into Ursina’s EditorCamera() class.

We can model the cube to be on a graph, where and a drag can be modeled as a start coordinate when the mouse was clicked and an end coordinate when the mouse was let go.

Rubik's Cube on cartisian set of axis




To make the left-click drag movements, the first step is to get the position of the mouse a of when the click started, then get the position b of when the click ended.

The pseudocode for this function is shown below.

DEF LEFT\_CLICK\_DRAG():

start = (x1, y1)

end = (x2, y2)

gradient = (y2 - y1) / (x2 - x1)

IF ABSOLUTE\_VALUE(gradient) > 1: # vertical line

IF x1 > 0 AND x2 > 0: # drag happens on right side of cube

IF y1 < y2: # drag is from low to high

rotate\_cube\_R()

ELSE:

rotate\_cube\_Ri()

ELSEIF x1 < 0 AND x2 < 0: # drag happens on left side of cube

IF y1 < y2:

rotate\_cube\_Li()

ELSE:

rotate\_cube\_Li()

ELSEIF ABSOLUTE\_VALUE(gradient) < 1: # horizontal line

IF y1 > 0 AND y2 > 0: # drag happens on upper side of cube

IF x1 < x2: # drag is from left to right

rotate\_cube\_Ui()

ELSE:

rotate\_cube\_U()

ELSEIF y1 < 0 AND y2 < 0: # drag happens on down side of cube

IF x1 < x2: # drag is from left to right

rotate\_cube\_D()

ELSE:

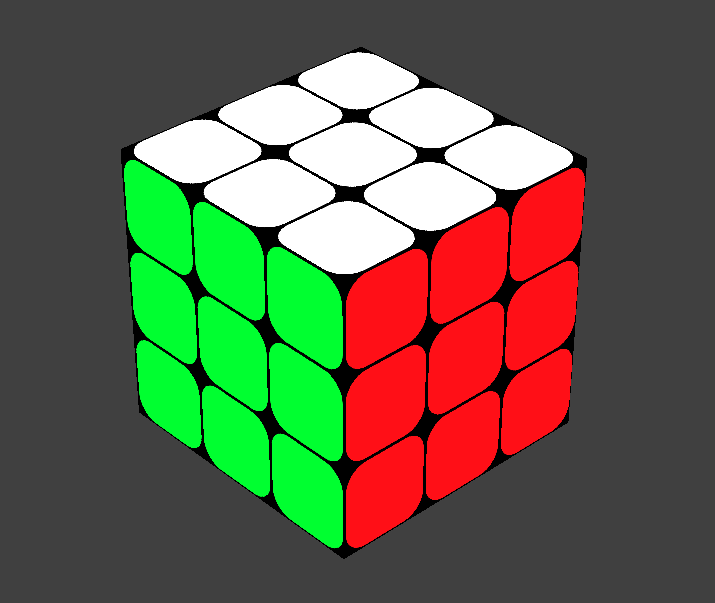


rotate\_cube\_Di()

Finding the front face

This code works only for the front face as when the camera is moved this code will rotate the wrong faces. To counteract this, we must figure out which face is the front face in relation to the camera. We can say that the distance from the center of the front face to the camara will shorter than the other faces. Now we need to change out model to a 3D set of axis’, where the Z increases as you go towards the back of t







Since the cubies are each 1 wide therefore have a radius of 0.5 from the center of the cubies, the distance from the center of the world to the center of the green middle facelet will be 1.5, or (0, 0, -1.5) on the 3D graph. And the camera position in this screen shot was (12.2095, 11.1038, -11.2974) we can use the 3D Pythagoras formula below, where the camera position is (x1, y1, z1) and the position of the facelet is (x2, y2, z1)



After calculating the distance to the camera from each face, shortest distance will be the front face. Now the last step is to change the rotations depending on which face is at the front, **for example, if the front face is green the R rotation = R rotation, however if the front face was green, the R rotation would be an F rotation on the green face, and the R rotation would be a B rotation** .

# Cube data structures

3D List

**This data structure stores facelets**

To store, read, and permutate the cube, I have used multiple date structures and method of representing the cube.

A picture containing rectangle

Description automatically generatedThe first method is a **3-Dimentional list**, of facelets, in rows in faces( cube[face][row][facelet], the facelet holds its color value as a string with the first letter of the color

Here, the facelet marked x will have the position cube[0][1][2] and will contain the string “W” for white.

The full 3D list for a completed cube is shown bellow.

self.cube = [

[['W', 'W', 'W'], # upper 0

['W', 'W', 'W'],

['W', 'W', 'W']],

[['G', 'G', 'G'], # front 1

['G', 'G', 'G'],

['G', 'G', 'G']],

[['R', 'R', 'R'], # right 2

['R', 'R', 'R'],

['R', 'R', 'R']],

[['O', 'O', 'O'], # left 3

['O', 'O', 'O'],

['O', 'O', 'O']],

[['Y', 'Y', 'Y'], # down 4

['Y', 'Y', 'Y'],

['Y', 'Y', 'Y']],

[['B', 'B', 'B'], # back 5

['B', 'B', 'B'],

['B', 'B', 'B']]

]

To rotate a face, a temporary variable holds a copy of the cube. To copy self.cube such that changing self.cube will not change the temp, the copy package is imported and copy.deepcopy(self.cube) is used.

All the face rotations are done by a moving a facelet to another position (its not pretty but there is no other way) as shown below in pseudocode which shows a rotation ( U )

self.cube[0][0][0], self.cube[0][0][1], self.cube[0][0][2], self.cube[0][1][0], self.cube[0][1][2], self.cube[0][2][0], self.cube[0][2][1], self.cube[0][2][2] =

self.cube[0][0][2], self.cube[0][1][2], self.cube[0][2][2], self.cube[0][0][1], self.cube[0][2][1], self.cube[0][0][0], self.cube[0][1][0], self.cube[0][2][0]

self.cube[1][0], self.cube[2][0], self.cube[5][0], self.cube[3][0] = temp[2][0], temp[5][0], temp[3][0], temp[1][0]

Cubie dictionaries

**This data structure stores cubies**

The second data structure used is a variable for each of the 26 cubies in the cube (not including the center cubie which has no colors). the cubies are named to describe their position in the cube, such that the up-front-right corner would be named “ufr”, 3 letter, one for each colored facelet on the cubie. The edge pieces only have 2 letters as they have 2 colored facelets, **for example, “fr” would describe the front right edge**. Finally, the center cubies only have one color, therefore one letter. The front center cubie’s variable is simply f.

Each cubie has an up, down, front, back, right, and left side but only 1,2 or 3 sides have facelets. Each cubie variable holds a **dictionary** containing a string with the first letter of a side as a key, and the position of the facelet in the 3D list as a key or an empty list if there is no facelet.

**For example**, the position of the ub (up right edge) cubie looks like this.

ub = {'u': cube[0][0][1], 'd': '', 'f': '', 'b': cube[5][0][1], 'r': '', 'l': ''}

To access the color of the upper facelet on the ub cubie, the query would be ub[‘u’]. I used this data structure primarily in solving algorithm because it makes the solution more intuitive.

Move history list

Each move performed on the cube is stored in a list variable with

# Solver algorithm

The solver algorithm solves the Rubik’s cube from one of any of its 43 quintillion permutations into its solved state. The solving method used is the beginner’s method because this program is made to teach beginners. While this method is not the most efficient, it is the easiest for a beginner to learn, hence the name. The methods below are described in more detail in the analysis section

## Solve White Cross (step 1)

The first step of solving the Rubik’s cube is to position the white edges in the correct place as shown below.

Rectangle

Description automatically generated

Solving the white cross is done mainly, find a white edge piece and move it into the correct location at the top where it matches the center facelet, there is no efficient way to do this with code other than the one shown below of going through every corner and insert.

Sides = [green, red, blue, orange]

For edge in sides

If top\_of\_edge1 == white AND left\_of\_edge1 == front face colour:

Perform move 1

Perform move 4

If top\_of\_edge2 == white AND left\_of\_edge2 == front face colour:

Perform move 5

Perform move 7

Repeat for each edge of the cube in each of its orientations

## Solve White Corners (step 2)

The next step of solving the Rubik’s cube is to put the white corners in the correct place, this completes the first layer of the cube.

Diagram

Description automatically generated

## Solve Second Layer (step 3)

A picture containing text, clipart

Description automatically generated

Diagram

Description automatically generated

## Solve Yellow Cross (step 4)

Diagram, schematic

Description automatically generated

## Match Yellow Edges (step 5

Diagram

Description automatically generated

## Position Yellow Corners (step 6)

Chart, diagram

Description automatically generated

## Orient Yellow Corners (step 7)

## Diagram Description automatically generated

# Webcam

This part of the program instructs the user to position their cube in front of the webcam 6 times, 1 for each face, the computer then works out what color each facelet of the cube is and inputs the colors into the 3D list.

A 3 by 3 grid is displayed on the screen along with an instruction of which face should the user show, paired with which face will be on the top. This ensures the user positions the right face in the right orientation in the right place. An example is shown below.

A picture containing indoor

Description automatically generated

After each picture is taken, the process of finding out the colors starts. For each facelet in a face, the code finds the color of 5 pixels from 5 coordinates around the center of the facelet using PIL.image.open(image\_file) and PIL.image.getpixel(coordinates) for each of the 5 coordinates.

Next, the average RGB value is calculated using numpy.mean(list\_of\_rgb\_vals, axis=0).

The average RGB values are then input into the 3D list of the cube. This is repeated each face of the cube.

The last step is to convert the RGB values in the cube into color names.

To ensure that the program works within a variety of different cube colors, light levels, and light hues, instead of setting values for each color such as red = (255,0,0) etc. The color’s value is set as the RGB value of the center piece with that color, as the center piece doesn’t move, we will always know what position it is at so we know its RGB value. **For example, the RGB value of red = the RGB value of the red center piece** . A dictionary containing a color and its RGB value for this specific case is made and for each facelet on the cube, the RGB value of the facelet is compared with the values of the dictionary to find the closest dictionary value, the RGB value is then replaced with the key of the closest value.

Finding the nearest color

Given an RGB value of unknown color, and a list of RGB values with their set colors, to find the color name, we can compare it to the list and see which color our unknown color is closest to. To do calculate the “distance” between two RGB values, we can model RGB as XYZ and use the 3D Pythagorean formula (or more formally the **Euclidian distance equation**) to find the distance, where (R1, G1, B1) = (X1, Y1, Z1) and (R2, G2, B2) = (X2, Y2, Z2)



After testing this method, I discovered that it cannot differentiate between white and yellow; red, orange and yellow; and green and blue. To solve this issue, I **converted the RGB values into HSV** (Hue, Saturation, Value). By finding the nearest hue value of the commonly switched up colors, I was able to drastically improve the accuracy of the code to determine the correct colours of the facelets.

**def** rgb\_to\_hsv(self, rgb): *# this function converts rgb values into hsv values* r, g, b = rgb  
 r, g, b = r / 255.0, g / 255.0, b / 255.0 *# converts rgb into a value in range 0-1* mx = max(r, g, b) *# find maximum and minimmum values* mn = min(r, g, b)  
 df = mx - mn  
 **if** mx == mn:  
 h = 0  
 **elif** mx == r:  
 h = (60 \* ((g - b) / df) + 360) % 360  
 **elif** mx == g:  
 h = (60 \* ((b - r) / df) + 120) % 360  
 **elif** mx == b:  
 h = (60 \* ((r - g) / df) + 240) % 360  
 **if** mx == 0:  
 s = 0  
 **else**:  
 s = (df / mx) \* 100  
 v = mx \* 100  
 hsv = (h,s,v)  
 **return** hsv

However, this method is not perfect. The code gets the colour wrong sometimes - which is unavoidable due to the varying lighting and reflections off the cube – so I added a feature that allows the user to edit the incorrect colours by entering a code. The edit code is described with Backus-Naur form as:

<face> = 0|1|2|3|4|5

<row> = 0|1|2

<facelet> = 0|1|2

<color> = “b”|”w”|”r”|”o”|”l”|”y”

<edit\_code> ::= <face><row><facelet><colour>

# Optimiser class

This class takes an input of the move history - a list of cube notation strings that describe a face or body rotation – and shortens the list to remove unnecessary terms so that the user can be given a solution for their cube that is as short as possible.

Unnecessary terms are:

1. A move and its inverse next to each other. E.g. R, Ri or Ui, U: these moves would cancel out to how no resultant effect on the cube
2. A move repeated twice. E.g. Li, Li: per standard cube notation, this is shortened to L2
3. A move repeated thrice. E.g. D, D, D: this is just the same as Di
4. A move repeated 4+ times: E.g. U, U, U, U, U: the first 4 result in no resultant rotation and any remainder is treated as normally.

Flatten

The first step of this shortening process is to expand all double terms (e.g. R2 becomes R, R ) this is done to make the string more easy to work with

Annihilate

Similarly to how when antimatter and mater interact, they annihilate, when a move and its inverse are next to each other, they too annihilate. This function goes through the entire list looking for these pairs and removing them. It repeats the run through until it runs through without editing a single term.

Run length Encoding

Run length encoding is typically used as a compression method to reduce the file size of plain text and images, however, it serves a different use in this case.

The RLE function changes a sequence of repeated elements into the element + the amount of times its repeated. In this case, the resultant rotation of a move repeated 4 or more times is the same as the remainder of the number of repetitions divided by 4; so the code saves the element + number of repetitions % 4 (% in python is the modulo operation).

Check letters

This function takes in the RLE list and changes a move repeated 3 times into the moves inverse.

# Hints

When the user is stuck and requests for help, the program will figure out which of the 7 steps the user is on or nearest to and provide them with the algorithm needed and how to apply them or solve the next step for the user.

Order of hints

If the hints are too obvious, the user will become reliant on them and not learn how to solve the cube. To counter this, the hint will start off vague and as the user asks for more hints, the hints will be more obvious, this diagram represents progression of hints.

A picture containing text, indoor, tiled

Description automatically generated

Identifying stage

To find out which stage the user is at. The program will check if the cube is complete then check stages 7 – 1 backwards until the current stage is found

# REMINDER single letter variable (underlined) is the centre cubies facelet colour

IF self.cube == self.completed\_cube:

Step = “completed”

.

.

ELSE if ur == {'u': W, 'd': '', 'f': r, 'b': '', 'r': '', 'l': ''} AND

ub == {'u': W, 'd': '', 'f': b, 'b': '', 'r': '', 'l': ''} AND

ul== {'u': W, 'd': '', 'f': l, 'b': '', 'r': '', 'l': ''} AND

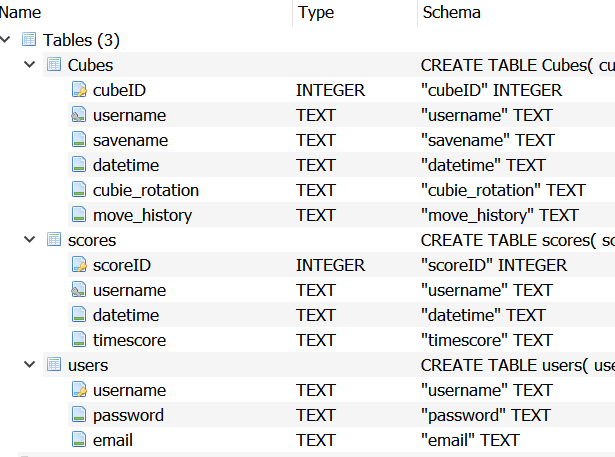
uf == {'u': W, 'd': '', 'f': f, 'b': '', 'r': '', 'l': ''}:

step = “white cross”

Database

In this project, I used a database to store user information, load and save cubes, and save scores.

Here is the layout of the database \*(  = private key ,  = foreign key)

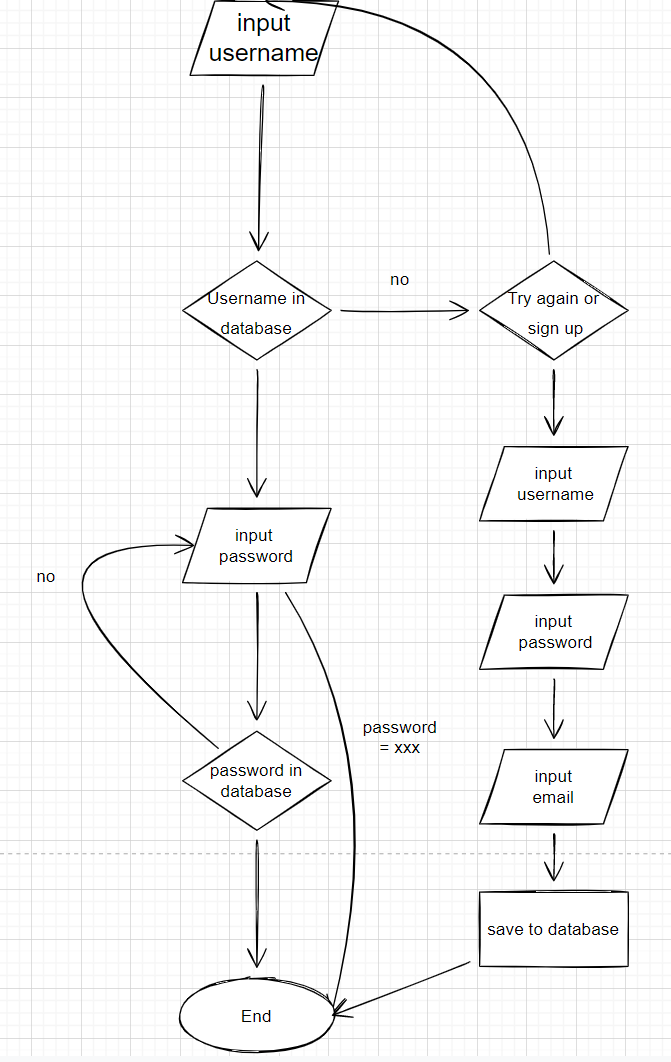


Saving cubes

To save cubes, the individual rotation of each cubie is saved to a list. To store this list in the database, I decided to use json to serialise the list and save it in a cell in the database as a string of text. Usually, it is ideal to atomise every cell in the database and achieve 3NF normalisation; however in this case, I believe keeping a list in the cell makes the database more simple and works better than creating a new table for each save.

Log in system

To make the log in system robust, I arranged it as shown in the flowchart below.



# Database functionalities

* User can sign up
* User can log in
* User can save the move history and position of a cube
* User can load previously saved cubes
* User can save their solve time
* User can have scoreboard displayed

Technical Solution

Rubiks\_database.py

import sqlite3  
  
class CubeDB():  
  
 def \_\_init\_\_(self):  
 self.logged\_in = False  
 self.username = **""** def create\_table(self, db\_name, sql\_txt):  
 with sqlite3.connect(db\_name) as db:  
 cursor = db.cursor()  
 cursor.execute(sql\_txt)  
 db.commit()  
  
 def create\_users\_table(self):  
 db\_name = **"Rubiks.db"** sqlCommand = **"""  
 CREATE TABLE users(  
 username TEXT,  
 password TEXT,  
 email TEXT,  
 primary key (username)  
 )  
 """** self.create\_table(db\_name, sqlCommand)  
  
  
 def create\_scores\_table(self):  
 db\_name = **"Rubiks.db"** sqlCommand = **"""  
 CREATE TABLE scores(  
 scoreID INTEGER,  
 username TEXT,  
 datetime TEXT,  
 timescore TEXT,  
 seconds INTEGER,  
 primary key (scoreID),  
 foreign key (username) references users (username)  
 )  
 """** self.create\_table(db\_name, sqlCommand)  
  
 def display\_scores(self):  
 with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 print(**"%-5s %15s %15s %12s"** % (**"ScoreID"**, **"Username"**, **"Score"**, **"date"**))  
 print(70 \* **"-"**)  
 for row in cursor.execute(**'SELECT \* FROM scores ORDER BY seconds'**):  
 scoreID, Username, date, score, seconds = row  
 print(**"%-5d %13s %16s %16s"** % (scoreID, Username, score, date[:10]))  
 conn.commit()  
  
 def create\_load\_cube\_table(self):  
 db\_name = **"Rubiks.db"** sqlCommand = **"""  
 CREATE TABLE cubes(  
 cubeID INTEGER,  
 username TEXT,  
 savename TEXT,  
 datetime TEXT,  
 cubie\_rotation TEXT,  
 move\_history TEXT,  
 primary key (cubeID),  
 foreign key (username) references users (username)  
 )  
 """** self.create\_table(db\_name, sqlCommand)  
  
 def sign\_up(self): *# allowes user to create an account* print(**"SIGN UP**\n**#################################"**)  
 username = input(**"Enter username"**)  
 password = input(**"Create a password"**)  
 email = input(**"Enter email"**)  
 with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 cursor.execute(**"INSERT INTO users VALUES (?,?,?)"**, [username, password, email])  
 conn.commit()  
 print(username, **": your account has been created"**)  
  
  
 def add\_score(self, values):  
 *# values being a tupple (username, datetime, timescore)* if self.logged\_in:  
  
 with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 cursor.execute(**"INSERT INTO scores (username, datetime, timescore, seconds) VALUES (?,?,?,?)"**, values)  
 conn.commit()  
  
 def add\_cube(self, values):  
 *# values being a tupple (username, savename, datetime, cubie\_rotation, move history)* with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 cursor.execute(**"INSERT INTO cubes (username, savename, datetime, cubie\_rotation, move\_history) VALUES (?,?,?,?,?)"**, values)  
 conn.commit()  
  
 def log\_in(self):  
 password = **""** username = input(**"Enter username:"**)  
 with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 cursor.execute(**"SELECT \* FROM users WHERE username = ?"**, [username])  
 record = cursor.fetchone()  
 if len(record) == 0:  
 option = str(input(**"user does not exist: Try again (1) or sign up (2)"**))  
 if option == **"1"**:  
 self.log\_in()  
 elif option == **"2"**:  
 self.sign\_up()  
 else:  
 while password.lower() != **"xxx"**:  
 password = input(**"Enter password:"**)  
 if password != record[1]:  
 print(**"incorrect password: enter 'xxx' to exit"**)  
 elif password == record[1]:  
 self.logged\_in = True  
 self.username = username  
 print(**"log in successful"**)  
 return  
 return False  
  
 def load\_cube(self):  
 if self.logged\_in:  
 with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 print(**"%5s %15s %15s"** % (**"cube ID"**, **"username"**, **"datetime"**))  
 print(45 \* **"-"**)  
 for row in cursor.execute(**'SELECT \* FROM cubes WHERE username = (?)'**, [self.username]):  
 cubeID, savename, datetime = row[0], row[1], row[2]  
 print(**"%5s %15s %23s"** % (cubeID, savename, datetime))  
 conn.commit()  
 cubeID = int(input(**"Enter Cube ID of the cube you want to load:"**))  
 cursor.execute(**'SELECT cubie\_rotation, move\_history FROM cubes WHERE cubeID = (?)'**, [cubeID])  
 loads = cursor.fetchone()  
 try:  
 return loads[0], loads[1]  
 except:  
 print(**"failed"**)  
 return 0,0  
 else:  
 print(**"you must log in first"**)  
 self.log\_in()  
 with sqlite3.connect(**"Rubiks.db"**) as conn:  
 cursor = conn.cursor()  
 print(**"%5s %15s %15s"** % (**"cube ID"**, **"username"**, **"datetime"**))  
 print(45 \* **"-"**)  
 for row in cursor.execute(**'SELECT \* FROM cubes WHERE username = (?)'**, [self.username]):  
 cubeID, savename, datetime = row[0], row[1], row[2]  
 print(**"%5s %15s %23s"** % (cubeID, savename, datetime))  
 conn.commit()  
 cubeID = int(input(**"Enter Cube ID of the cube you want to load:"**))  
 cursor.execute(**'SELECT cubie\_rotation, move\_history FROM cubes WHERE cubeID = (?)'**, [cubeID])  
 loads = cursor.fetchone()  
 return loads[0], loads[1]  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 dbinst = CubeDB()  
 try:  
 dbinst.create\_users\_table()  
 dbinst.create\_scores\_table()  
 dbinst.create\_load\_cube\_table()  
 except sqlite3.OperationalError:  
 pass

Main

from ursina import \*  
import copy  
import sys  
import time  
import random  
import numpy as np  
import json  
import datetime  
import cv2 as cv  
from PIL import Image  
from rubiks\_database import CubeDB  
  
*# VARIABLES  
#############################################################*app = Ursina()  
s = Sky()  
window.size = (800, 800)  
window.borderless = False  
window.fullscreen = False  
window.position = (50, 50)  
EditorCamera() *# adds camera controls to ursina scene*Text.size = 0.02  
Text.default\_resolution = 1080 \* Text.size  
options\_txt = Text(background=True, x=-0.8, y=0.4,  
 text=**"Reset: 1 Save: 2 Solve next level: 3 Scramble: 4 Solve all: 5 Load cube: 6 Input cube: 7"**)  
state\_txt = Text(visible=False, background=True, x=0.3, y=0.4, text=**"next step = solve white cross"**)  
timer\_txt = Text(size=0.04, origin=(0, -7), background=True, text=**"0:00"**, center=True)  
  
tip\_txt = Text(x=-0.8, y=-0.4,  
 text=**"HOLD 'Z' to run through chosen solution: TAP SPACEBAR to begin and end solve timer: PRESS ESCAPE FOR MENU"**,  
 color=color.red)  
  
*# Test saved cube*saved = [(0, 180, -90), (0, 0, -90), (-90, -90, -90), (0, 180, -90), (0, 0, 90), (90, 0, -90), (0, 90, -90),  
 (-90, 0, 0), (0, 0, 0), (0, 0, -90), (0, 0, -90), (0, 0, -90), (0, -180, 90), (0, -180, -90), (0, -180, 0),  
 (-90, 0, 90), (0, -180, -90), (0, -90, -180), (0, 90, 0), (90, 0, 0), (90, 0, 0), (-90, 0, -90), (0, 0, 90),  
 (0, 90, 180), (90, -90, 90), (0, -90, -90), (90, 0, 180)]  
  
*# Test saved solution and scramble*example = [**'F'**, **'U'**, **'D'**, **'Bi'**, **'D'**, **'Di'**, **'U'**, **'D'**, **'L'**, **'Ui'**, **'D'**, **'D'**, **'F'**, **'Ri'**, **'Fi'**, **'D'**, **'D'**, **'Bi'**, **'D'**, **'D'**,  
 **'Li'**, **'Ui'**, **'L'**, **'Ri'**, **'U'**, **'U'**, **'R'**, **'Ui'**, **'Bi'**, **'U'**, **'B'**, **'U'**, **'L'**, **'Ui'**, **'Li'**, **'U'**, **'U'**, **'R'**, **'Ui'**, **'Ri'**,  
 **'Ui'**, **'Fi'**, **'U'**, **'F'**, **'L'**, **'Fi'**, **'Li'**, **'F'**, **'Li'**, **'Ui'**, **'L'**, **'L'**, **'Fi'**, **'Li'**, **'F'**, **'Li'**, **'Ui'**, **'L'**, **'R'**,  
 **'Bi'**, **'Ri'**, **'B'**, **'Ri'**, **'Ui'**, **'R'**, **'U'**, **'L'**, **'Fi'**, **'Li'**, **'F'**, **'Li'**, **'Ui'**, **'L'**, **'Bi'**, **'R'**, **'B'**, **'Ri'**, **'B'**, **'U'**,  
 **'Bi'**, **'U'**, **'U'**, **'F'**, **'U'**, **'R'**, **'Ui'**, **'Ri'**, **'Fi'**, **'U'**, **'Ri'**, **'F'**, **'R'**, **'Bi'**, **'Ri'**, **'Fi'**, **'R'**, **'B'**, **'U'**, **'U'**,  
 **'Ri'**, **'Ui'**, **'R'**, **'U'**, **'Di'**, **'R'**, **'R'**, **'U'**, **'Ri'**, **'U'**, **'R'**, **'Ui'**, **'R'**, **'Ui'**, **'R'**, **'R'**, **'D'**, **'U'**, **'U'**]  
  
*# dictionary of information of different turns format of dictionary is axis, layer, rotation*rot\_dict = {**'u'**: [**'y'**, 1, 90], **'d'**: [**'y'**, -1, -90], **'l'**: [**'x'**, -1, -90], **'r'**: [**'x'**, 1, 90], **'f'**: [**'z'**, -1, 90],  
 **'b'**: [**'z'**, 1, -90]}  
  
*# variables for input function*inst3D = None  
instDB = None  
movelist = []  
t = 0  
prev = 0  
current = 0  
vector = [2, 3]  
counter = 0  
running = False  
menu\_up = False  
menu\_instance = None  
hint\_count = 0  
  
timer = 0.0  
timer\_on = False  
*# build in ursina function, it is called every time a key is pressed*hint\_count = 0  
  
  
def hints():  
 *# create instance of solver class* instSolve = Solver()  
 *# put 3d cube pos into solver.cube by running through the list of moves that the 3d cube has done* instSolve.runthrough(inst3D.movelist3D)  
 *# display the current completion stage of the 3d cube* solvestate = instSolve.check\_state()  
 state\_txt.text = solvestate[0] *# current solved state* state\_txt.visible = True *# lets user see current state*hint\_button = Button(text=**"Hints"**, text\_color=color.red, text\_scale=2, origin=(7, 0), color=color.blue,  
 highlight\_color=color.azure, pressed\_color=color.green, scale=0.1, on\_click=hints  
  
)  
  
  
def secs\_to\_mins(secs):  
 mins = secs / 60  
 secs = round((mins - int(mins)) \* 60)  
 if secs < 10:  
 secs = **"0"** + str(secs)  
  
 mins\_txt = **f"**{int(mins)}**:**{secs}**"** return mins\_txt  
  
  
def update():  
 global timer  
 time\_elapsed = int(time.time() - timer)  
 formatted\_time = secs\_to\_mins(time\_elapsed)  
 if timer\_txt.text != formatted\_time and timer\_on:  
 timer\_txt.text = formatted\_time  
  
  
def input(key):  
 global t, prev, current, vector, counter, running, menu\_up, menu\_instance, example, timer, timer\_on  
  
 if key == **"space"**:  
 if timer\_on:  
 timer\_on = False  
 time\_elapsed = int(time.time() - timer)  
 timer\_txt.color = color.white  
 print((**"do you wish to save your time score? y/n"**))  
 answer = sys.stdin.readline()[:1].lower()  
 if answer == **"y"**:  
 if not instDB.logged\_in:  
 print(**"you must log in first"**)  
 instDB.log\_in()  
 instDB.add\_score(  
 [instDB.username, str(datetime.datetime.now())[:16], secs\_to\_mins(time\_elapsed), time\_elapsed])  
 print(**"score added"**)  
 else:  
 pass  
 timer\_txt.text = **"0:00"** elif not timer\_on:  
 timer\_on = True  
 timer = time.time()  
 timer\_txt.color = color.red  
 if key == **"j"**:  
 print(LVector3f(camera.world\_position))  
 print(LVector3f(camera.world\_rotation))  
 print((mouse.position.x \* 10, mouse.position.y \* 10, mouse.position.z \* 10), **"**\n**"**)  
 if key == **"1"**:  
 inst3D.reset()  
 *# save the cube* if key == **"2"**:  
 inst3D.save()  
  
 *# solve layer* if key == **"3"**:  
 *# create instance of solver class* instSolve = Solver()  
 *# put 3d cube pos into solver.cube by running through the list of moves that the 3d cube has done* instSolve.runthrough(inst3D.movelist3D)  
 *# display the current completion stage of the 3d cube* solvestate = instSolve.check\_state()  
 state\_txt.text = solvestate[0] *# current solved state* state\_txt.visible = True *# lets user see current state  
 # saves moves to solve next step* nextstep = solvestate[1].replace(**" "**, **""**) *# next step function string is created* instSolve.show = True *# appends moves to solve list but doesnt perform them or add them to move list* if nextstep != **"Solved"**: *# cannot solve already solved cube* exec(**f"instSolve.**{nextstep}**()"**) *# perform next step* else:  
 return  
 instSolve.show = False *# next step didn't change instsolve.cube* example = instSolve.solvelist *# list of moves to solve cube* example = Optimiser(example).flatten(example)  
 print(example)  
 counter = 0  
 *# scramble* if key == **"9"**:  
 inst3D.build\_cube()  
 if held\_keys[**"4"**] and not time.time() - t < 0.3:  
 inst3D.rotator(random.choice(list(rot\_dict.keys())) + random.choice([**""**, **"i"**]), speed=0.2)  
 t = time.time()  
 *# solve all* if key == **"5"**:  
 instSolve = Solver() *# create instance of solver class* instSolve.runthrough(inst3D.movelist3D) *# put 3d cube pos into solver.cube* instSolve.show = True  
 instSolve.solveCube()  
 instSolve.show = False  
 example = instSolve.solvelist *# list of moves to solve cube* example = Optimiser(example).flatten(example)  
 print(**"3d inputs this"**)  
 instSolve.printCube()  
 print(example)  
 *# load cube* if key == **"6"**:  
 *# inst3D.delete\_all()* inst3D.load()  
 *# open menu* if key == **"7"**:  
 if hint\_count == 0:  
 *# create instance of solver class* instSolve = Solver()  
 *# put 3d cube pos into solver.cube by running through the list of moves that the 3d cube has done* instSolve.runthrough(inst3D.movelist3D)  
 *# display the current completion stage of the 3d cube* solvestate = instSolve.check\_state()  
 state\_txt.text = solvestate[0] *# current solved state* state\_txt.visible = True *# lets user see current state* elif hint\_count == 1:  
 pass  
 if key == **"8"**:  
 input\_cube()  
  
 if key == **"escape"**:  
 if not menu\_up:  
 menu\_instance = MainMenu()  
 menu\_instance.display\_menu(menu\_instance.main\_menu)  
 menu\_up = True  
 else:  
 menu\_instance = None  
 menu\_up = False  
  
 *# mouse input* if key == **"left mouse down"**:  
 current = 1  
 else:  
 current = 0  
 if prev == 0 and current == 1:  
 vector[0] = LVector3f(mouse.position).x, LVector3f(mouse.position).y  
 prev = 1  
 if prev == 1 and current == 0:  
 vector[1] = LVector3f(mouse.position).x, LVector3f(mouse.position).y  
 prev = 0  
 if not time.time() - t < 0.5:  
 inst3D.gradient(vector[0], vector[1])  
 t = time.time()  
 *# run sequence* if held\_keys[**"z"**]:  
 running = True  
  
 if counter == len(example):  
 running = False  
 inst3D.USD = False  
  
 if not time.time() - t < 0.3 and running:  
 inst3D.rotator(example[counter], speed=0.1)  
 t = time.time()  
 counter += 1  
  
 *# time keeper, prevents spam* elif time.time() - t < 0.6 or key not in rot\_dict: *# doesnt allow user to spam buttons* return  
  
 *# does rotation of key press* else:  
 t = time.time()  
 inst3D.rotator(inst3D.adjustment(inst3D.get\_front\_face())[key.upper()])  
  
  
def input\_cube():  
 print(**"in"**)  
 global inst3D  
 instCV = CubeCV()  
 instCV.webcam()  
 print(**"out"**)  
 if instCV.check\_legitimacy():  
 inst3D = Cube3D(cube=instCV.cube)  
 inst3D.build\_cube(instCV.cube\_to\_rotations())  
 inst3D.initCube()  
 app.run()  
 else:  
 print(**"Capture failed, try in different lighting"**)  
  
  
class Cube3D():  
 def \_\_init\_\_(self, cube=[]):  
 self.cube = cube  
 self.center = Entity() *# sets center as the center of rotation* self.movelist3D = []  
 self.runfront = **"F"** self.col\_to\_face\_dict = {**"G"**: **"F"**, **"B"**: **"B"**, **"O"**: **"L"**, **"R"**: **"R"**, **"U"**: **"Y"**, **"D"**: **"D"**}  
 self.USD = False  
  
 *# this function creates the cube entities and puts them in a list* def initCube(self):  
 if len(self.cube) == 0:  
 for x in (-1, 0, 1): *# these nested for loops create 27 cubie entities* for y in (-1, 0, 1):  
 for z in (-1, 0, 1):  
 pos = (x, y, z)  
 self.cube.append(  
 Entity(model=**'cube\_model.obj'**, texture=**'cube\_texture.png'**, position=pos, rotation=(0, 0, 0),  
 scale=0.5)) *# entity defines and creates the 27 cubes,  
  
 # deletes all cube entities* def reset(self):  
 self.delete\_all()  
 self.initCube()  
 self.movelist3D = []  
  
 def delete\_all(self):  
 if len(self.cube) > 0:  
 for i in reversed(self.cube):  
 destroy(i)  
 self.cube.clear()  
  
 *# attaches cubies on a face to the center* def parent\_child\_relation(self, axis, layer):  
 *# parent relates to the middle of a face and the children are the 8 outer cubies* for c in self.cube:  
 c.position, c.rotation = round(c.world\_position, 1), c.world\_rotation  
 c.parent = scene  
  
 self.center.rotation = 0  
  
 for c in self.cube:  
  
 if eval(**f'c.position.**{axis}**'**) == layer:  
 c.parent = self.center  
  
 def get\_front\_face(self):  
 camera\_position = LVector3f(camera.world\_position) *# xyz position of where camera is* center\_location = {**"U"**: (0, 1, 0), **"D"**: (0, -1, 0), **"L"**: (-1, 0, 0), **"R"**: (1, 0, 0), **"F"**: (0, 0, -1),  
 **"B"**: (0, 0, 1)} *# location of each og the center cubies* distance\_dict = {}  
 for key, value in center\_location.items(): *# calculates distance between centers and camera* distance = sqrt((value[0] - camera\_position[0]) \*\* 2 + (value[1] - camera\_position[1]) \*\* 2 + (  
 value[2] - camera\_position[2]) \*\* 2)  
 distance\_dict[round(distance, 2)] = key  
  
 front\_face = distance\_dict[min(distance\_dict)]  
 return front\_face *# returns nearest face as string* def adjustment(self, front\_face):  
 *# sets which corresponding face you must turn depending on which face is at the front* adjustment\_dict = {**"F"**: **"F"**, **"R"**: **"R"**, **"B"**: **"B"**, **"L"**: **"L"**, **"U"**: **"U"**, **"D"**: **"D"**}  
 if front\_face == **"F"**:  
 adjustment\_dict = {**"F"**: **"F"**, **"R"**: **"R"**, **"B"**: **"B"**, **"L"**: **"L"**, **"U"**: **"U"**, **"D"**: **"D"**}  
 elif front\_face == **"R"**:  
 adjustment\_dict = {**"F"**: **"R"**, **"R"**: **"B"**, **"B"**: **"L"**, **"L"**: **"F"**, **"U"**: **"U"**, **"D"**: **"D"**}  
 elif front\_face == **"L"**:  
 adjustment\_dict = {**"F"**: **"L"**, **"R"**: **"F"**, **"B"**: **"R"**, **"L"**: **"B"**, **"U"**: **"U"**, **"D"**: **"D"**}  
 elif front\_face == **"B"**:  
 adjustment\_dict = {**"F"**: **"B"**, **"R"**: **"L"**, **"B"**: **"F"**, **"L"**: **"R"**, **"U"**: **"U"**, **"D"**: **"D"**}  
 elif front\_face == **"U"**:  
 adjustment\_dict = {**"F"**: **"U"**, **"R"**: **"R"**, **"B"**: **"D"**, **"L"**: **"L"**, **"U"**: **"B"**, **"D"**: **"F"**}  
 elif front\_face == **"D"**:  
 adjustment\_dict = {**"F"**: **"D"**, **"R"**: **"R"**, **"B"**: **"U"**, **"L"**: **"L"**, **"U"**: **"F"**, **"D"**: **"B"**}  
 return adjustment\_dict  
  
 *# sets which corresponding face you must turn depending on if the cube is upside down* def upside\_down(self, move):  
 adjustment\_dict = {**"F"**: **"F"**, **"R"**: **"L"**, **"B"**: **"B"**, **"L"**: **"R"**, **"U"**: **"D"**, **"D"**: **"U"**}  
 adjusted\_move = adjustment\_dict[move[0]]  
 return adjusted\_move  
  
 *# performs move based on mouse movement* def gradient(self, start, end):  
 x1, y1 = start  
 x2, y2 = end  
 if sqrt((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) < 0.05:  
 return  
 try: *# if the gradient becomes infinite put it s 100* grad = (y2 - y1) / (x2 - x1)  
 except ZeroDivisionError:  
 grad = 100  
 front\_face = self.get\_front\_face()  
 adjustment\_dict = self.adjustment(front\_face)  
 if abs(grad) > 1: *# if vertical drag* if x1 > 0 and x2 > 0: *# if drag occurs on right side of cube* if y1 < y2: *# if drag is from bottom to top* self.rotator(adjustment\_dict[**"R"**])  
 else:  
 self.rotator(adjustment\_dict[**"R"**] + **"i"**)  
 elif x1 < 0 and x2: *# if drag occurs on left side of cube* if y1 < y2:  
 self.rotator(adjustment\_dict[**"L"**] + **"i"**)  
 else:  
 self.rotator(adjustment\_dict[**"L"**])  
 elif abs(grad) < 1: *# if horizontal drag* if y1 > 0 and y2 > 0: *# if drag occurs on top side of cube* if x1 < x2: *# if drag is from right to left* self.rotator(adjustment\_dict[**"U"**] + **"i"**)  
 else:  
 self.rotator(adjustment\_dict[**"U"**])  
  
 elif y1 < 0 and y2 < 0:  
 if x1 < x2:  
 self.rotator(adjustment\_dict[**"D"**])  
 else:  
 self.rotator(adjustment\_dict[**"D"**] + **"i"**)  
  
 def get\_up\_face(self, front):  
 *# call this when down or up is at the front.  
 # find the closest of these to the camera position and apply the previously defined adjusts to it* possible\_degrees = [-180, -90, 0, 90, 180]  
  
 *# when looking at bot yellow,  
 # up = green = -90,0,0  
 # up = red = -90,-90,0  
 # up = blue = -90,0,-180  
 # up = orange = -90,-90,-180  
  
 # when looking at top white,  
 # up = green = 90,0,0  
 # up = red = 90,-90,180  
 # up = blue = 90,-180,180  
 # up = orange = 90,90,0* pass *# find vector to get to each cubeie, the greatest gradient will be the top cubie  
  
 # rotates side of cube* def rotator(self, key, speed=0.4):  
 key = key.upper()  
 *# if a turn move:* if key[0] == **"Y"**:  
 *# new color of front = key[-1]* self.runfront = self.col\_to\_face\_dict[key[-1]]  
 return  
 if key[0] == **"Z"**:  
 *# turn cube upside down if a z move is done* if self.USD:  
 self.USD = False  
 else:  
 self.USD = True  
 return  
 if held\_keys[**"z"**]:  
 if self.USD: *# if cube is upside down, adjust the move accordingly* if len(key) == 1: *# if clockwise rotation* key = self.upside\_down(key)  
 else: *# if anticlockwise rotation* key = self.upside\_down(key) + key[-1]  
 if running:  
 if len(key) == 1:  
 key = self.adjustment(self.runfront)[key[0]]  
 else:  
 key = self.adjustment(self.runfront)[key[0]] + key[-1]  
  
 key = key.lower()  
 axis, layer, rotation = rot\_dict[key[0]] *# e.g 'x', 1, 90* self.parent\_child\_relation(axis, layer)  
 *# if shift is helpd, perform the inverse move* shift = held\_keys[**"shift"**]  
 if shift:  
 self.movelist3D.append(key.upper() + **"i"**)  
 elif len(key) == 1:  
 self.movelist3D.append(key.upper())  
 else:  
 self.movelist3D.append(key[0].upper() + **"i"**)  
  
 if shift or key[-1] == **"i"**:  
 rotation = -rotation  
 eval(**f'self.center.animate\_rotation\_**{axis} **(**{rotation}**, duration =** {speed}**)'**)  
  
 *# loads saved cube* def load(self):  
 global instDB, inst3D  
  
 cubie\_rotations, move\_history = instDB.load\_cube()  
 if cubie\_rotations == 0:  
 return  
 cubie\_rotations = json.loads(cubie\_rotations)  
 inst3D.movelist3D = json.loads(move\_history)  
 self.build\_cube(cubie\_rotations)  
 print(**"Cube loaded"**)  
  
 def build\_cube(self, cubie\_rotations=saved):  
  
 self.delete\_all()  
 *# inst3D.cube.clear()* inst3D.cube = []  
 for x in range(-1, 2):  
 for y in range(-1, 2):  
 for z in range(-1, 2):  
 pos = (x, y, z)  
 inst3D.cube.append(Entity(model=**'cube\_model.obj'**, texture=**'cube\_texture.png'**, position=pos,  
 rotation=cubie\_rotations.pop(), scale=0.5))  
  
 *# saves position of cubes* def save(self):  
 global saved  
 while 1:  
  
 if not instDB.logged\_in:  
 print(**"you must log in first to save a cube"**)  
 instDB.log\_in()  
  
 rotations = []  
 for i in reversed(range(27)):  
 d = self.cube[i]  
 i, j, k = d.rotation.x, d.rotation.y, d.rotation.z  
 rotations.append((int(i), int(j), int(k)))  
 saved = rotations  
 savename = input2(**"Enter save name:"**)  
 instDB.add\_cube([instDB.username, savename, str(datetime.datetime.now())[:16], json.dumps(saved),  
 json.dumps(inst3D.movelist3D)])  
 print(**f"'**{savename}**' saved"**)  
 break  
 **"""else:  
 print("you must log in first to save a cube")  
 if not instDB.log\_in():  
 print("not saved")  
 return"""**class Optimiser:  
 def \_\_init\_\_(self, start\_lst):  
 self.start\_lst = start\_lst  
 *# start\_list = ["R", "U", "R2", "R3", "D2", "U1"]  
 # self.start\_lst = self.make\_list("F F F D R R' R", prime\_val="'")* self.flattened = self.flatten(self.start\_lst)  
 self.annihilated = self.annihilate(self.flattened.copy())  
 self.rle = self.run\_length\_encode(self.annihilated)  
 self.checked = self.check\_letters(self.rle)  
  
 def check\_letters(self, input\_lst):  
 output\_lst = []  
 *# loop through the letters in the input list* for letter in input\_lst:  
 number = letter[-1]  
 *# if there is a number on the end of the move (the move is repeated)* if number.isnumeric():  
 letters, number = letter[:-1], int(number)  
 *# if there are 3 of a move on a run, replace it with 1 of the inverse move* if number == 3:  
 if **"i"** in letters:  
 letters = letters.replace(**"i"**, **""**)  
 else:  
 letters += **"i"** output\_lst.append(letters)  
 *# if there are 2 moves on a run, always display it with the basic move (rather than the inverse) because  
 # they are the same* elif number == 2:  
 output\_lst.append(letters.replace(**"i"**, **""**) + **"2"**)  
 *# otherwise, just add the move to the list as it was* else:  
 output\_lst.append(letters + str(number))  
 *# otherwise, add the move with no number attached* else:  
 output\_lst.append(letter)  
  
 return output\_lst  
  
 def run\_length\_encode(self, input\_lst):  
 counter = 1  
 letter = **""** output = []  
 *# loops through the moves in the list* for new\_letter in input\_lst:  
 *# if the new letter is the same as the old letter, add 1 to the counter for that letter* if new\_letter == letter:  
 counter += 1  
 else:  
 *# if the new letter is not the first letter, add the old letter and its counter to the list* if letter != **""**:  
 *# use %4 because doing 4 of the same move on a run goes all the way round the cube* counter = counter % 4  
 *# if the counter is not 0, add the letter to the list* if counter > 0:  
 *# only add the number to the move if it is more than 1* num = str(counter) if counter > 1 else **""** output.append(letter + num)  
 letter = new\_letter  
 counter = 1  
 *# add the final letter and its number to the list, in the same way as it was done in the loop* counter = counter % 4  
 if counter > 0:  
 num = str(counter) if counter > 1 else **""** output.append(letter + num)  
  
 return output  
  
 def flatten(self, input\_lst):  
 output\_lst = []  
 *# loop through the items in the input list* for item in input\_lst:  
 *# get the number corresponding to how many of the moves are done on a run* number = item[-1]  
 *# if the 'number' is a letter, there is no number so there must be only one of the move done* if number.isnumeric() and item[0] != **"Z"**:  
 *# get the move that is being done and append it to the list 'number' times* move = item[:-1]  
 for \_ in range(int(number)):  
 output\_lst.append(move)  
 else:  
 output\_lst.append(item)  
  
 return output\_lst  
  
 def annihilate(self, input\_lst):  
 index = 0  
 letter = **""** letters\_changed = True  
 *# loop round until there have been no changes (similar to bubble sort lol)* while letters\_changed:  
 *# loop through the list with a variable counter so that you can remove items and change the counter back to  
 # where the new items are.* letters\_changed = False  
 while index < len(input\_lst):  
 new\_letter = input\_lst[index]  
 remove\_val = False  
 *# check for move then inverse* if new\_letter == letter + **"i"** or new\_letter + **"i"** == letter:  
 remove\_val = True  
  
 *# remove the letter and the letter before and move the counter back to the old letter* if remove\_val:  
 del input\_lst[index]  
 del input\_lst[index - 1]  
 index -= 2  
 letters\_changed = True  
 letter = **""** *# increment the counter* else:  
 index += 1  
 letter = new\_letter  
  
 return input\_lst  
  
  
class MenuButton(Button):  
  
 def \_\_init\_\_(self, text=None, \*\*kwargs):  
 super().\_\_init\_\_(text, scale=(.25, .075), highlight\_color=color.azure, \*\*kwargs)  
  
 for key, value in kwargs.items():  
 setattr(self, key, value)  
  
  
class MainMenu:  
  
 def \_\_init\_\_(self):  
 self.button\_spacing = .075 \* 1.25  
 self.menu\_parent = Entity(parent=camera.ui, y=.15)  
  
 self.no\_menu = Entity(parent=self.menu\_parent)  
 self.main\_menu = Entity(parent=self.menu\_parent)  
 self.load\_menu = Entity(parent=self.menu\_parent)  
 self.options\_menu = Entity(parent=self.menu\_parent)  
  
 self.style\_menu = Entity(parent=self.menu\_parent)  
 self.control\_menu = Entity(parent=self.menu\_parent)  
 state\_handler = Animator(  
 {**'main\_menu'**: self.main\_menu, **'load\_menu'**: self.load\_menu, **'options\_menu'**: self.options\_menu,  
 **'no\_menu'**: self.no\_menu, **'style\_menu'**: self.style\_menu, **'control\_menu'**: self.control\_menu})  
  
 self.main\_menu.buttons = [MenuButton(**'resume'**, on\_click=Func(setattr, state\_handler, **'state'**, **'no\_menu'**)),  
 *# MenuButton('reset', on\_click=Func(inst3D.reset)),* MenuButton(**'scoreboard'**, on\_click=Func(CubeDB().display\_scores)),  
 MenuButton(**'save game'**, on\_click=Func(inst3D.save)),  
 MenuButton(**'load game'**, on\_click=Func(inst3D.load)),  
 MenuButton(**'log in'**, on\_click=Func(instDB.log\_in)),  
 MenuButton(**'options'**, on\_click=Func(setattr, state\_handler, **'state'**, **'options\_menu'**)),  
 MenuButton(**'quit'**, on\_click=Sequence(Wait(.01), Func(sys.exit)))]  
  
 self.options\_menu.buttons = [  
  
 MenuButton(parent=self.options\_menu, text=**"style"**, y=-1 \* self.button\_spacing,  
 on\_click=Func(setattr, state\_handler, **'state'**, **'style\_menu'**)),  
 MenuButton(parent=self.options\_menu, text=**"controls"**, y=-2 \* self.button\_spacing,  
 on\_click=Func(setattr, state\_handler, **'state'**, **'control\_menu'**)),  
 *# MenuButton(parent=self.options\_menu, text="", y=-3 \* self.button\_spacing),* MenuButton(parent=self.options\_menu, text=**'back'**, y=(-5 \* self.button\_spacing),  
 on\_click=Func(setattr, state\_handler, **'state'**, **'main\_menu'**))]  
  
 self.style\_menu.buttons = [MenuButton(parent=self.style\_menu, text=**"pastel"**, y=-1 \* self.button\_spacing),  
 MenuButton(parent=self.style\_menu, text=**"original"**, y=-2 \* self.button\_spacing),  
 MenuButton(parent=self.style\_menu, text=**"colour blind"**, y=-3 \* self.button\_spacing),  
 MenuButton(parent=self.style\_menu, text=**'back'**, y=(-5 \* self.button\_spacing),  
 on\_click=Func(setattr, state\_handler, **'state'**, **'options\_menu'**))]  
  
 self.control\_menu.buttons = [MenuButton(parent=self.control\_menu, text=**"RL UD FB"**, y=-1 \* self.button\_spacing),  
 MenuButton(parent=self.control\_menu, text=**"DA WS QE"**, y=-2 \* self.button\_spacing),  
 MenuButton(parent=self.control\_menu, text=**'back'**, y=(-5 \* self.button\_spacing),  
 on\_click=Func(setattr, state\_handler, **'state'**, **'options\_menu'**))]  
  
 def display\_menu(self, menu):  
 for count, entity in enumerate(menu.buttons):  
 entity.parent = menu  
 entity.y = -count \* self.button\_spacing  
  
  
class handler:  
 def \_\_init\_\_(self, cube=None):  
 if cube is None:  
 cube = [[[**'W'**, **'W'**, **'W'**], *# upper 0* [**'W'**, **'W'**, **'W'**], [**'W'**, **'W'**, **'W'**]],  
  
 [[**'G'**, **'G'**, **'G'**], *# front 1* [**'G'**, **'G'**, **'G'**], [**'G'**, **'G'**, **'G'**]],  
  
 [[**'R'**, **'R'**, **'R'**], *# right 2* [**'R'**, **'R'**, **'R'**], [**'R'**, **'R'**, **'R'**]],  
  
 [[**'O'**, **'O'**, **'O'**], *# left 3* [**'O'**, **'O'**, **'O'**], [**'O'**, **'O'**, **'O'**]],  
  
 [[**'Y'**, **'Y'**, **'Y'**], *# down 4* [**'Y'**, **'Y'**, **'Y'**], [**'Y'**, **'Y'**, **'Y'**]],  
  
 [[**'B'**, **'B'**, **'B'**], *# back 5* [**'B'**, **'B'**, **'B'**], [**'B'**, **'B'**, **'B'**]]  
  
 ]  
 self.show = False  
 self.solvelist = []  
 self.movelist = []  
 *# cube[face][row][column]* self.completed\_cube = [[[**'W'**, **'W'**, **'W'**], *# upper 0* [**'W'**, **'W'**, **'W'**], [**'W'**, **'W'**, **'W'**]],  
  
 [[**'G'**, **'G'**, **'G'**], *# front 1* [**'G'**, **'G'**, **'G'**], [**'G'**, **'G'**, **'G'**]],  
  
 [[**'R'**, **'R'**, **'R'**], *# right 2* [**'R'**, **'R'**, **'R'**], [**'R'**, **'R'**, **'R'**]],  
  
 [[**'O'**, **'O'**, **'O'**], *# left 3* [**'O'**, **'O'**, **'O'**], [**'O'**, **'O'**, **'O'**]],  
  
 [[**'Y'**, **'Y'**, **'Y'**], *# down 4* [**'Y'**, **'Y'**, **'Y'**], [**'Y'**, **'Y'**, **'Y'**]],  
  
 [[**'B'**, **'B'**, **'B'**], *# back 5* [**'B'**, **'B'**, **'B'**], [**'B'**, **'B'**, **'B'**]]  
  
 ]  
 self.cube = cube  
 self.d = **'Y'** self.u = **'W'** self.f = **'G'** self.b = **'B'** self.r = **'R'** self.l = **'O'** self.uf = {**'u'**: self.cube[0][2][1], **'d'**: **''**, **'f'**: self.cube[1][0][1], **'b'**: **''**, **'r'**: **''**, **'l'**: **''**}  
 self.ur = {**'u'**: self.cube[0][1][2], **'d'**: **''**, **'f'**: **''**, **'b'**: **''**, **'r'**: self.cube[2][0][1], **'l'**: **''**}  
 self.ub = {**'u'**: self.cube[0][0][1], **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][0][1], **'r'**: **''**, **'l'**: **''**}  
 self.ul = {**'u'**: self.cube[0][1][0], **'d'**: **''**, **'f'**: **''**, **'b'**: **''**, **'r'**: **''**, **'l'**: self.cube[3][0][1]}  
 self.df = {**'u'**: **''**, **'d'**: self.cube[4][0][1], **'f'**: self.cube[1][2][1], **'b'**: **''**, **'r'**: **''**, **'l'**: **''**}  
 self.dr = {**'u'**: **''**, **'d'**: self.cube[4][1][2], **'f'**: **''**, **'b'**: **''**, **'r'**: self.cube[2][2][1], **'l'**: **''**}  
 self.db = {**'u'**: **''**, **'d'**: self.cube[4][2][1], **'f'**: **''**, **'b'**: self.cube[5][2][1], **'r'**: **''**, **'l'**: **''**}  
 self.dl = {**'u'**: **''**, **'d'**: self.cube[4][1][0], **'f'**: **''**, **'b'**: **''**, **'r'**: **''**, **'l'**: self.cube[3][2][1]}  
 self.fr = {**'u'**: **''**, **'d'**: **''**, **'f'**: self.cube[1][1][2], **'b'**: **''**, **'r'**: self.cube[2][1][0], **'l'**: **''**}  
 self.fl = {**'u'**: **''**, **'d'**: **''**, **'f'**: self.cube[1][1][0], **'b'**: **''**, **'r'**: **''**, **'l'**: self.cube[3][1][2]}  
 self.br = {**'u'**: **''**, **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][1][0], **'r'**: self.cube[2][1][2], **'l'**: **''**}  
 self.bl = {**'u'**: **''**, **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][1][2], **'r'**: **''**, **'l'**: self.cube[3][1][0]}  
  
 self.ufr = {**'u'**: self.cube[0][2][2], **'d'**: **''**, **'f'**: self.cube[1][0][2], **'b'**: **''**, **'r'**: self.cube[2][0][0],  
 **'l'**: **''**}  
 self.ufl = {**'u'**: self.cube[0][2][0], **'d'**: **''**, **'f'**: self.cube[1][0][0], **'b'**: **''**, **'r'**: **''**,  
 **'l'**: self.cube[3][0][2]}  
 self.ubr = {**'u'**: self.cube[0][0][2], **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][0][0], **'r'**: self.cube[2][0][2],  
 **'l'**: **''**}  
 self.ubl = {**'u'**: self.cube[0][0][0], **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][0][2], **'r'**: **''**,  
 **'l'**: self.cube[3][0][0]}  
 self.dfr = {**'u'**: **''**, **'d'**: self.cube[4][0][2], **'f'**: self.cube[1][2][2], **'b'**: **''**, **'r'**: self.cube[2][2][0],  
 **'l'**: **''**}  
 self.dfl = {**'u'**: **''**, **'d'**: self.cube[4][0][0], **'f'**: self.cube[1][2][0], **'b'**: **''**, **'r'**: **''**,  
 **'l'**: self.cube[3][2][2]}  
 self.dbr = {**'u'**: **''**, **'d'**: self.cube[4][2][2], **'f'**: **''**, **'b'**: self.cube[5][2][0], **'r'**: self.cube[2][2][2],  
 **'l'**: **''**}  
 self.dbl = {**'u'**: **''**, **'d'**: self.cube[4][2][0], **'f'**: **''**, **'b'**: self.cube[5][2][2], **'r'**: **''**,  
 **'l'**: self.cube[3][2][0]}  
  
 self.movedict = {**"U"**: self.U(), **"Ui"**: self.Ui(), **"D"**: self.D(), **"Di"**: self.Di(), **"R"**: self.R(), **"Ri"**: self.Ri(),  
 **"L"**: self.L(), **"Li"**: self.Li(), **"F"**: self.F(), **"Fi"**: self.Fi(), **"B"**: self.B(), **"Bi"**: self.Bi()}  
  
 *# updates cubies* def update\_cubies(self):  
 self.uf = {**'u'**: self.cube[0][2][1], **'d'**: **''**, **'f'**: self.cube[1][0][1], **'b'**: **''**, **'r'**: **''**, **'l'**: **''**}  
 self.ur = {**'u'**: self.cube[0][1][2], **'d'**: **''**, **'f'**: **''**, **'b'**: **''**, **'r'**: self.cube[2][0][1], **'l'**: **''**}  
 self.ub = {**'u'**: self.cube[0][0][1], **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][0][1], **'r'**: **''**, **'l'**: **''**}  
 self.ul = {**'u'**: self.cube[0][1][0], **'d'**: **''**, **'f'**: **''**, **'b'**: **''**, **'r'**: **''**, **'l'**: self.cube[3][0][1]}  
 self.df = {**'u'**: **''**, **'d'**: self.cube[4][0][1], **'f'**: self.cube[1][2][1], **'b'**: **''**, **'r'**: **''**, **'l'**: **''**}  
 self.dr = {**'u'**: **''**, **'d'**: self.cube[4][1][2], **'f'**: **''**, **'b'**: **''**, **'r'**: self.cube[2][2][1], **'l'**: **''**}  
 self.db = {**'u'**: **''**, **'d'**: self.cube[4][2][1], **'f'**: **''**, **'b'**: self.cube[5][2][1], **'r'**: **''**, **'l'**: **''**}  
 self.dl = {**'u'**: **''**, **'d'**: self.cube[4][1][0], **'f'**: **''**, **'b'**: **''**, **'r'**: **''**, **'l'**: self.cube[3][2][1]}  
 self.fr = {**'u'**: **''**, **'d'**: **''**, **'f'**: self.cube[1][1][2], **'b'**: **''**, **'r'**: self.cube[2][1][0], **'l'**: **''**}  
 self.fl = {**'u'**: **''**, **'d'**: **''**, **'f'**: self.cube[1][1][0], **'b'**: **''**, **'r'**: **''**, **'l'**: self.cube[3][1][2]}  
 self.br = {**'u'**: **''**, **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][1][0], **'r'**: self.cube[2][1][2], **'l'**: **''**}  
 self.bl = {**'u'**: **''**, **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][1][2], **'r'**: **''**, **'l'**: self.cube[3][1][0]}  
  
 self.ufr = {**'u'**: self.cube[0][2][2], **'d'**: **''**, **'f'**: self.cube[1][0][2], **'b'**: **''**, **'r'**: self.cube[2][0][0],  
 **'l'**: **''**}  
 self.ufl = {**'u'**: self.cube[0][2][0], **'d'**: **''**, **'f'**: self.cube[1][0][0], **'b'**: **''**, **'r'**: **''**,  
 **'l'**: self.cube[3][0][2]}  
 self.ubr = {**'u'**: self.cube[0][0][2], **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][0][0], **'r'**: self.cube[2][0][2],  
 **'l'**: **''**}  
 self.ubl = {**'u'**: self.cube[0][0][0], **'d'**: **''**, **'f'**: **''**, **'b'**: self.cube[5][0][2], **'r'**: **''**,  
 **'l'**: self.cube[3][0][0]}  
 self.dfr = {**'u'**: **''**, **'d'**: self.cube[4][0][2], **'f'**: self.cube[1][2][2], **'b'**: **''**, **'r'**: self.cube[2][2][0],  
 **'l'**: **''**}  
 self.dfl = {**'u'**: **''**, **'d'**: self.cube[4][0][0], **'f'**: self.cube[1][2][0], **'b'**: **''**, **'r'**: **''**,  
 **'l'**: self.cube[3][2][2]}  
 self.dbr = {**'u'**: **''**, **'d'**: self.cube[4][2][2], **'f'**: **''**, **'b'**: self.cube[5][2][0], **'r'**: self.cube[2][2][2],  
 **'l'**: **''**}  
 self.dbl = {**'u'**: **''**, **'d'**: self.cube[4][2][0], **'f'**: **''**, **'b'**: self.cube[5][2][2], **'r'**: **''**,  
 **'l'**: self.cube[3][2][0]}  
  
 def scramble(self):  
 for \_ in range(20):  
 self.convertmoves(random.choice(list(self.movedict.keys())))  
  
 *# displays cube in nice format* def printCube(self):  
 self.update\_cubies()  
 print(**"**\n\t**"** + self.ubl[**'u'**] + self.ub[**'u'**] + self.ubr[**'u'**] + **"**\n\t**"** + self.ul[**'u'**] + self.u + self.ur[  
 **'u'**] + **"**\n\t**"** + self.ufl[**'u'**] + self.uf[**'u'**] + self.ufr[**'u'**] + **"**\n**"** + \  
 \  
 self.ubl[**'l'**] + self.ul[**'l'**] + self.ufl[**'l'**] + **" "** + self.ufl[**'f'**] + self.uf[**'f'**] + self.ufr[**'f'**] + **" "** +  
 self.ufr[**'r'**] + self.ur[**'r'**] + self.ubr[**'r'**] + **" "** + self.ubr[**'b'**] + self.ub[**'b'**] + self.ubl[**'b'**] + **"**\n**"** + \  
 self.bl[**'l'**] + self.l + self.fl[**'l'**] + **" "** + self.fl[**'f'**] + self.f + self.fr[**'f'**] + **" "** + self.fr[  
 **'r'**] + self.r + self.br[**'r'**] + **" "** + self.br[**'b'**] + self.b + self.bl[**'b'**] + **" "** + **"**\n**"** + self.dbl[  
 **'l'**] + self.dl[**'l'**] + self.dfl[**'l'**] + **" "** + self.dfl[**'f'**] + self.df[**'f'**] + self.dfr[**'f'**] + **" "** +  
 self.dfr[**'r'**] + self.dr[**'r'**] + self.dbr[**'r'**] + **" "** + self.dbr[**'b'**] + self.db[**'b'**] + self.dbl[  
 **'b'**] + **"**\n\t**"** + \  
 \  
 self.dfl[**'d'**] + self.df[**'d'**] + self.dfr[**'d'**] + **"**\n\t**"** + self.dl[**'d'**] + self.d + self.dr[**'d'**] + **"**\n\t**"** +  
 self.dbl[**'d'**] + self.db[**'d'**] + self.dbr[**'d'**] + **"**\n**"**)  
  
 print(**"====================================="**)  
  
 *# displays cube in nice format* def display\_cube(self):  
 for i in self.cube:  
 print()  
 for j in i:  
 print(j)  
 print(**"====================================="**)  
  
 *# runs through sequence* def runthrough(self, sequence):  
 self.cube = copy.deepcopy(self.completed\_cube)  
 for move in sequence:  
 self.convertmoves(move)  
  
 *# Finds put the rotation of each cubie in a list* def cube\_to\_rotations(self):  
 rotations = []  
 order\_list = [**"dfl"**, **"dl"**, **"dbl"**, **"fl"**, **"l"**, **"bl"**, **"ufl"**, **"ul"**, **"ubl"**, **"df"**, **"d"**, **"db"**, **"f"**, **"center"**, **"b"**,  
 **"uf"**, **"u"**, **"ub"**, **"dfr"**, **"dr"**, **"dbr"**, **"fr"**, **"r"**, **"br"**, **"ufr"**, **"ur"**, **"ubr"**]  
 possible\_dicts = [*# (X,Y,Z)  
 # (R,U,F)  
 # W TOP* ({**'u'**: **'W'**, **'d'**: **'Y'**, **'f'**: **'G'**, **'b'**: **'B'**, **'r'**: **'R'**, **'l'**: **'O'**}, (0, 0, 0)),  
 ({**'u'**: **'W'**, **'d'**: **'Y'**, **'f'**: **'R'**, **'b'**: **'O'**, **'r'**: **'B'**, **'l'**: **'G'**}, (0, 90, 0)),  
 ({**'u'**: **'W'**, **'d'**: **'Y'**, **'f'**: **'B'**, **'b'**: **'G'**, **'r'**: **'O'**, **'l'**: **'R'**}, (0, 180, 0)),  
 ({**'u'**: **'W'**, **'d'**: **'Y'**, **'f'**: **'O'**, **'b'**: **'R'**, **'r'**: **'G'**, **'l'**: **'B'**}, (0, -90, 0)), *# Y TOP* ({**'u'**: **'Y'**, **'d'**: **'W'**, **'f'**: **'G'**, **'b'**: **'B'**, **'r'**: **'O'**, **'l'**: **'R'**}, (0, 0, 180)),  
 ({**'u'**: **'Y'**, **'d'**: **'W'**, **'f'**: **'O'**, **'b'**: **'R'**, **'r'**: **'B'**, **'l'**: **'G'**}, (0, 90, 180)),  
 ({**'u'**: **'Y'**, **'d'**: **'W'**, **'f'**: **'B'**, **'b'**: **'G'**, **'r'**: **'R'**, **'l'**: **'O'**}, (0, 180, 180)),  
 ({**'u'**: **'Y'**, **'d'**: **'W'**, **'f'**: **'R'**, **'b'**: **'O'**, **'r'**: **'G'**, **'l'**: **'B'**}, (0, -90, 180)), *# R TOP* ({**'u'**: **'R'**, **'d'**: **'O'**, **'f'**: **'G'**, **'b'**: **'B'**, **'r'**: **'Y'**, **'l'**: **'W'**}, (0, 0, -90)),  
 ({**'u'**: **'R'**, **'d'**: **'O'**, **'f'**: **'Y'**, **'b'**: **'W'**, **'r'**: **'B'**, **'l'**: **'G'**}, (0, 90, -90)),  
 ({**'u'**: **'R'**, **'d'**: **'O'**, **'f'**: **'B'**, **'b'**: **'G'**, **'r'**: **'W'**, **'l'**: **'Y'**}, (0, 180, -90)),  
 ({**'u'**: **'R'**, **'d'**: **'O'**, **'f'**: **'W'**, **'b'**: **'Y'**, **'r'**: **'G'**, **'l'**: **'B'**}, (0, -90, -90)), *# O TOP* ({**'u'**: **'O'**, **'d'**: **'R'**, **'f'**: **'G'**, **'b'**: **'B'**, **'r'**: **'W'**, **'l'**: **'Y'**}, (0, 0, 90)),  
 ({**'u'**: **'O'**, **'d'**: **'R'**, **'f'**: **'W'**, **'b'**: **'Y'**, **'r'**: **'B'**, **'l'**: **'G'**}, (0, 90, 90)),  
 ({**'u'**: **'O'**, **'d'**: **'R'**, **'f'**: **'B'**, **'b'**: **'G'**, **'r'**: **'Y'**, **'l'**: **'W'**}, (0, 180, 90)),  
 ({**'u'**: **'O'**, **'d'**: **'R'**, **'f'**: **'Y'**, **'b'**: **'W'**, **'r'**: **'G'**, **'l'**: **'B'**}, (0, -90, 90)), *# G TOP* ({**'u'**: **'G'**, **'d'**: **'B'**, **'f'**: **'Y'**, **'b'**: **'W'**, **'r'**: **'R'**, **'l'**: **'O'**}, (90, 0, 0)),  
 ({**'u'**: **'G'**, **'d'**: **'B'**, **'f'**: **'R'**, **'b'**: **'O'**, **'r'**: **'W'**, **'l'**: **'Y'**}, (90, 0, 90)),  
 ({**'u'**: **'G'**, **'d'**: **'B'**, **'f'**: **'W'**, **'b'**: **'Y'**, **'r'**: **'O'**, **'l'**: **'R'**}, (90, 0, 180)),  
 ({**'u'**: **'G'**, **'d'**: **'B'**, **'f'**: **'O'**, **'b'**: **'R'**, **'r'**: **'Y'**, **'l'**: **'W'**}, (90, 0, -90)), *# B TOP* ({**'u'**: **'B'**, **'d'**: **'G'**, **'f'**: **'W'**, **'b'**: **'Y'**, **'r'**: **'R'**, **'l'**: **'O'**}, (-90, 0, 0)),  
 ({**'u'**: **'B'**, **'d'**: **'G'**, **'f'**: **'R'**, **'b'**: **'O'**, **'r'**: **'Y'**, **'l'**: **'W'**}, (-90, 0, -90)),  
 ({**'u'**: **'B'**, **'d'**: **'G'**, **'f'**: **'Y'**, **'b'**: **'W'**, **'r'**: **'O'**, **'l'**: **'R'**}, (-90, 0, 180)),  
 ({**'u'**: **'B'**, **'d'**: **'G'**, **'f'**: **'O'**, **'b'**: **'R'**, **'r'**: **'W'**, **'l'**: **'Y'**}, (-90, 0, 90))]  
  
 self.update\_cubies()  
 for cubie in order\_list:  
 for key, rot in possible\_dicts:  
 if len(cubie) == 3:  
 *# if cube is a corner and matches this case, append the rotation value* if eval(**f"self.**{cubie}**['**{cubie[0]}**'] ==** {key}**['**{cubie[0]}**'] and "  
 f"self.**{cubie}**['**{cubie[1]}**'] ==** {key}**['**{cubie[1]}**'] and "  
 f"self.**{cubie}**['**{cubie[2]}**'] ==** {key}**['**{cubie[2]}**']"**):  
 rotations.append(rot)  
  
 elif len(cubie) == 2:  
 *# if cube is an edge and matches this case, append the rotation value* if eval(**f"self.**{cubie}**['**{cubie[0]}**'] ==** {key}**['**{cubie[0]}**'] and "  
 f"self.**{cubie}**['**{cubie[1]}**'] ==** {key}**['**{cubie[1]}**']"**):  
 rotations.append(rot)  
 elif len(cubie) == 1 or cubie == **"center"**:  
 rotations.append((0, 0, 0))  
 break  
 *# self.printCube()  
 # self.display\_cube()* rotations.reverse()  
 return rotations  
  
 *# Given a string, performs calls the rotation function* def convertmoves(self, move):  
 self.movelist.append(move)  
  
 if self.show: *# if we are just creating a solve sequence without change ing the actual cube, we use this to append the move to a list without changeing anythong real* self.solvelist.append(move) *# return* if move == **"Y"**:  
 self.Y()  
 elif move == **"Yi"**:  
 self.Yi()  
 elif move == **"Z2"**:  
 self.Z2()  
 elif move == **"U2"**:  
 self.U2()  
 elif move == **"D2"**:  
 self.D2()  
 elif move == **"R2"**:  
 self.R2()  
 elif move == **"L2"**:  
 self.L2()  
 elif move == **"F2"**:  
 self.F2()  
 elif move == **"B2"**:  
 self.B2()  
 elif move == **"U"**:  
 self.U()  
 elif move == **"Ui"**:  
 self.Ui()  
 elif move == **"D"**:  
 self.D()  
 elif move == **"Di"**:  
 self.Di()  
 elif move == **"R"**:  
 self.R()  
 elif move == **"Ri"**:  
 self.Ri()  
 elif move == **"L"**:  
 self.L()  
 elif move == **"Li"**:  
 self.Li() *#* elif move == **"F"**:  
 self.F()  
 elif move == **"Fi"**:  
 self.Fi()  
 elif move == **"B"**:  
 self.B()  
 elif move == **"Bi"**:  
 self.Bi()  
 else:  
 pass  
 self.update\_cubies()  
  
 if move in [**"Y"**, **"Yi"**]:  
 try:  
 self.solvelist[-1] = self.solvelist[-1] + self.f  
 except IndexError:  
 pass  
  
 def rotate(self, side, clockwise=True):  
 face = self.cube[side]  
  
 if not clockwise:  
 face[0][0], face[0][1], face[0][2], face[1][0], face[1][2], face[2][0], face[2][1], face[2][2] = face[0][2], \  
 face[1][2], \  
 face[2][2], \  
 face[0][1], \  
 face[2][1], \  
 face[0][0], \  
 face[1][0], \  
 face[2][0]  
  
 self.cube[0][0][0], self.cube[0][0][1], self.cube[0][0][2], self.cube[0][1][0], self.cube[0][1][2], \  
 self.cube[0][2][0], self.cube[0][2][1], self.cube[0][2][2] = self.cube[0][0][2], self.cube[0][1][2], \  
 self.cube[0][2][2], self.cube[0][0][1], \  
 self.cube[0][2][1], self.cube[0][0][0], \  
 self.cube[0][1][0], self.cube[0][2][0]  
  
  
 else:  
 face[0][2], face[1][2], face[2][2], face[0][1], face[2][1], face[0][0], face[1][0], face[2][0] = face[0][0], \  
 face[0][1], \  
 face[0][2], \  
 face[1][0], \  
 face[1][2], \  
 face[2][0], \  
 face[2][1], \  
 face[2][2]  
  
 def U(self):  
 temp = copy.deepcopy(self.cube)  
 self.rotate(0, True)  
 self.cube[1][0], self.cube[2][0], self.cube[5][0], self.cube[3][0] = temp[2][0], temp[5][0], temp[3][0], \  
 temp[1][0]  
  
 def Ui(self):  
 temp = copy.deepcopy(self.cube)  
 self.rotate(0, False)  
 self.cube[2][0], self.cube[5][0], self.cube[3][0], self.cube[1][0] = temp[1][0], temp[2][0], temp[5][0], \  
 temp[3][0]  
  
 def D(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[1][2], self.cube[2][2], self.cube[5][2], self.cube[3][2] = temp[3][2], temp[1][2], temp[2][2], \  
 temp[5][2]  
 self.rotate(4)  
  
 def Di(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[3][2], self.cube[1][2], self.cube[2][2], self.cube[5][2] = temp[1][2], temp[2][2], temp[5][2], \  
 temp[3][2]  
 self.rotate(4, False)  
  
 def R(self):  
 temp = copy.deepcopy(self.cube)  
 self.rotate(2, True)  
 self.cube[0][0][2], self.cube[0][1][2], self.cube[0][2][2] = temp[1][0][2], temp[1][1][2], temp[1][2][2]  
 self.cube[1][0][2], self.cube[1][1][2], self.cube[1][2][2] = temp[4][0][2], temp[4][1][2], temp[4][2][2]  
 self.cube[4][0][2], self.cube[4][1][2], self.cube[4][2][2] = temp[5][2][0], temp[5][1][0], temp[5][0][0]  
 self.cube[5][0][0], self.cube[5][1][0], self.cube[5][2][0] = temp[0][2][2], temp[0][1][2], temp[0][0][2]  
  
 def Ri(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[1][0][2], self.cube[1][1][2], self.cube[1][2][2] = temp[0][0][2], temp[0][1][2], temp[0][2][2]  
 self.cube[4][0][2], self.cube[4][1][2], self.cube[4][2][2] = temp[1][0][2], temp[1][1][2], temp[1][2][2]  
 self.cube[5][2][0], self.cube[5][1][0], self.cube[5][0][0] = temp[4][0][2], temp[4][1][2], temp[4][2][2]  
 self.cube[0][2][2], self.cube[0][1][2], self.cube[0][0][2] = temp[5][0][0], temp[5][1][0], temp[5][2][0]  
  
 self.rotate(2, False)  
  
 def L(self):  
 temp = copy.deepcopy(self.cube)  
 self.rotate(3)  
 self.cube[1][0][0], self.cube[1][1][0], self.cube[1][2][0] = temp[0][0][0], temp[0][1][0], temp[0][2][0]  
 self.cube[4][0][0], self.cube[4][1][0], self.cube[4][2][0] = temp[1][0][0], temp[1][1][0], temp[1][2][0]  
 self.cube[5][0][2], self.cube[5][1][2], self.cube[5][2][2] = temp[4][2][0], temp[4][1][0], temp[4][0][0]  
 self.cube[0][0][0], self.cube[0][1][0], self.cube[0][2][0] = temp[5][2][2], temp[5][1][2], temp[5][0][2]  
  
 def Li(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[0][0][0], self.cube[0][1][0], self.cube[0][2][0] = temp[1][0][0], temp[1][1][0], temp[1][2][0]  
 self.cube[1][0][0], self.cube[1][1][0], self.cube[1][2][0] = temp[4][0][0], temp[4][1][0], temp[4][2][0]  
 self.cube[4][2][0], self.cube[4][1][0], self.cube[4][0][0] = temp[5][0][2], temp[5][1][2], temp[5][2][2]  
 self.cube[5][2][2], self.cube[5][1][2], self.cube[5][0][2] = temp[0][0][0], temp[0][1][0], temp[0][2][0]  
  
 self.rotate(3, False)  
  
 def F(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[3][0][2], self.cube[3][1][2], self.cube[3][2][2] = temp[4][0]  
 self.cube[0][2] = [temp[3][2][2], temp[3][1][2], temp[3][0][2]]  
 self.cube[2][0][0], self.cube[2][1][0], self.cube[2][2][0] = temp[0][2]  
 self.cube[4][0] = [temp[2][2][0], temp[2][1][0], temp[2][0][0]]  
  
 self.rotate(1, True)  
  
 def Fi(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[4][0] = [temp[3][0][2], temp[3][1][2], temp[3][2][2]]  
 self.cube[3][2][2], self.cube[3][1][2], self.cube[3][0][2] = temp[0][2]  
 self.cube[0][2] = [temp[2][0][0], temp[2][1][0], temp[2][2][0]]  
 self.cube[2][2][0], self.cube[2][1][0], self.cube[2][0][0] = temp[4][0]  
  
 self.rotate(1, False)  
  
 def B(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[2][2][2], self.cube[2][1][2], self.cube[2][0][2] = temp[4][2]  
 self.cube[0][0] = [temp[2][0][2], temp[2][1][2], temp[2][2][2]]  
 self.cube[3][2][0], self.cube[3][1][0], self.cube[3][0][0] = temp[0][0]  
 self.cube[4][2] = [temp[3][0][0], temp[3][1][0], temp[3][2][0]]  
  
 self.rotate(5, True)  
  
 def Bi(self):  
 temp = copy.deepcopy(self.cube)  
 self.cube[4][2] = [temp[2][2][2], temp[2][1][2], temp[2][0][2]]  
 self.cube[2][0][2], self.cube[2][1][2], self.cube[2][2][2] = temp[0][0]  
 self.cube[0][0] = [temp[3][2][0], temp[3][1][0], temp[3][0][0]]  
 self.cube[3][0][0], self.cube[3][1][0], self.cube[3][2][0] = temp[4][2]  
  
 self.rotate(5, False)  
  
 def U2(self):  
 self.U()  
 self.U()  
  
 def D2(self):  
 self.D()  
 self.D()  
  
 def R2(self):  
 self.R()  
 self.R()  
  
 def L2(self):  
 self.L()  
 self.L()  
  
 def F2(self):  
 self.F()  
 self.F()  
  
 def B2(self):  
 self.B()  
 self.B()  
  
 def Y(self):  
 self.cube[1], self.cube[2], self.cube[5], self.cube[3] = self.cube[2], self.cube[5], self.cube[3], self.cube[1]  
 self.rotate(0)  
 self.rotate(4, False)  
 self.f, self.r, self.b, self.l = self.r, self.b, self.l, self.f  
  
 def Yi(self):  
 self.cube[2], self.cube[5], self.cube[3], self.cube[1] = self.cube[1], self.cube[2], self.cube[5], self.cube[3]  
 self.rotate(0, False)  
 self.rotate(4)  
 self.r, self.b, self.l, self.f = self.f, self.r, self.b, self.l  
  
 def Z2(self):  
 self.cube[0], self.cube[4], self.cube[2], self.cube[3] = self.cube[4], self.cube[0], self.cube[3], self.cube[2]  
 self.u, self.d, self.r, self.l = self.d, self.u, self.l, self.r  
 self.update\_cubies()  
 for i in range(len(self.cube)):  
 self.rotate(i)  
 self.rotate(i)  
  
  
*# CUBE SOLVER  
########################################################################################################################*class Solver(handler):  
 def \_\_init\_\_(self, cube=handler().completed\_cube):  
 handler.\_\_init\_\_(self, cube=cube)  
 self.srt = lambda piece: sorted(**''**.join(list(piece.values())))  
  
 *# returns whether the the top layer and the middle cubie match color, and whether the top face has a cross on it* def cross(self, yface): *# yface is face on the y axis, color, so white or yellow* cross = True  
 matched = True  
 for face in [**"b"**, **"r"**, **"f"**, **"l"**]:  
 if not eval(**f"self.d**{face}**['**{face}**'] == self.**{face}**"**):  
 matched = False  
 if not eval(**f"self.d**{face}**['d'] == '**{yface}**'"**):  
 cross = False  
 return cross, matched  
  
 *# checks the current stage of completion the cube is solved to* def check\_state(self):  
  
 if self.cube == self.completed\_cube:  
 return [**"Solved"**, **"Solved"**]  
  
 f2l = True *# first 2 layers* f1l = True *# first 1 layer* if not self.cube[0] == self.completed\_cube[0]: *# if top face isnt complete, f2l and f1l cannot be true* f2l = False  
 f1l = False  
 for row in [0, 1]:  
 for face in [1, 2, 3, 5]:  
 if not (self.cube[face][row] == self.completed\_cube[face][row]): *# row is not solved* f2l = False  
 if row == 0: *# if the first row is not solved* f1l = False  
  
 cross, matched = self.cross(**"Y"**)  
  
 self.convertmoves(**"Z2"**) *# turns upside down then back to normal so that we can use the check done function* cd = self.checkdone() *# true if all corners are in correct position* self.convertmoves(**"Z2"**)  
 if matched and cross and f2l and cd:  
 return [**"Yellow corners Positioned"**, **"orient Yellow Corners"**]  
 if matched and cross and f2l:  
 return [**"Yellow edges matched"**, **"position Yellow Corners"**]  
  
 if cross and f2l:  
 return [**"Yellow cross completed"**, **"match Yellow Edges"**]  
  
 if f2l:  
 return [**"Second layer completed"**, **"solve Yellow Cross"**]  
 self.convertmoves(**"Z2"**) *# flip to check for white cross* cross, matched = self.cross(**"W"**)  
 self.convertmoves(**"Z2"**)  
 *# print(cross)* if cross and f1l:  
 return [**"White corners solved"**, **"solve Second Layer"**]  
  
 if cross and matched:  
 return [**"White cross solved"**, **"solve White Corners"**]  
 else:  
 return [**"Cube not at any stage"**, **"solve White Cross"**]  
  
 *# if self.cube[0] == self.completed\_cube[0] and  
  
 # functions solve step of the cube* def solveWhiteCross(self):  
 self.update\_cubies()  
 sides = [**'G'**, **'R'**, **'B'**, **'O'**]  
 for side in sides:  
 if self.dr[**'d'**] == **'W'** and self.dr[**'r'**] == side:  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.db[**'d'**] == **'W'** and self.db[**'b'**] == side:  
 self.convertmoves(**"B"**)  
 self.convertmoves(**"B"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.dl[**'d'**] == **'W'** and self.dl[**'l'**] == side:  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"Ui"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.fr[**'f'**] == **'W'** and self.fr[**'r'**] == side:  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.fl[**'f'**] == **'W'** and self.fl[**'l'**] == side:  
 self.convertmoves(**"Li"**)  
 self.convertmoves(**"Ui"**)  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.br[**'b'**] == **'W'** and self.br[**'r'**] == side:  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.bl[**'b'**] == **'W'** and self.bl[**'l'**] == side:  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"Ui"**)  
 self.convertmoves(**"Li"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.uf[**'u'**] == **'W'** and self.uf[**'f'**] == side:  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.ur[**'u'**] == **'W'** and self.ur[**'r'**] == side:  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.ul[**'u'**] == **'W'** and self.ul[**'l'**] == side:  
 self.convertmoves(**"Ui"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.ub[**'u'**] == **'W'** and self.ub[**'b'**] == side:  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.dr[**'d'**] == side and self.dr[**'r'**] == **'W'**:  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"F"**)  
  
 elif self.db[**'d'**] == side and self.db[**'b'**] == **'W'**:  
 self.convertmoves(**"B"**)  
 self.convertmoves(**"D"**)  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"Di"**)  
  
 elif self.dl[**'d'**] == side and self.dl[**'l'**] == **'W'**:  
 self.convertmoves(**"Li"**)  
 self.convertmoves(**"Fi"**)  
  
 elif self.fr[**'f'**] == side and self.fr[**'r'**] == **'W'**:  
 self.convertmoves(**"F"**)  
  
 elif self.fl[**'f'**] == side and self.fl[**'l'**] == **'W'**:  
 self.convertmoves(**"Fi"**)  
  
 elif self.br[**'b'**] == side and self.br[**'r'**] == **'W'**:  
 self.convertmoves(**"B"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"Bi"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.bl[**'b'**] == side and self.bl[**'l'**] == **'W'**:  
 self.convertmoves(**"Bi"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"B"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"F"**)  
  
 elif self.uf[**'u'**] == side and self.uf[**'f'**] == **'W'**:  
 self.convertmoves(**"Ui"**)  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"R"**)  
  
 elif self.ur[**'u'**] == side and self.ur[**'r'**] == **'W'**:  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"R"**)  
  
 elif self.ul[**'u'**] == side and self.ul[**'l'**] == **'W'**:  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"Fi"**)  
 self.convertmoves(**"Li"**)  
  
 elif self.ub[**'u'**] == side and self.ub[**'b'**] == **'W'**:  
 self.convertmoves(**"U"**)  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"R"**)  
  
 elif self.df[**'d'**] == side and self.df[**'f'**] == **'W'**:  
 self.convertmoves(**"Fi"**)  
 self.convertmoves(**"D"**)  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"Di"**)  
  
 self.convertmoves(**"Y"**)  
  
 self.convertmoves(**"F2"**)  
 self.convertmoves(**"R2"**)  
 self.convertmoves(**"B2"**)  
 self.convertmoves(**"L2"**)  
  
 def solveWhiteCorners(self):  
 self.update\_cubies()  
 white\_corners = [**"WGR"**, **"WRB"**, **"WBO"**, **"WOG"**]  
 for corner in white\_corners:  
 *# Moves corner to* if self.ufl[**'u'**] in corner and self.ufl[**'f'**] in corner and self.ufl[**'l'**] in corner:  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"D"**)  
 self.convertmoves(**"Li"**)  
 elif self.ubl[**'u'**] in corner and self.ubl[**'b'**] in corner and self.ubl[**'l'**] in corner:  
 self.convertmoves(**"Li"**)  
 self.convertmoves(**"D2"**)  
 self.convertmoves(**"L"**)  
 elif self.ubr[**'u'**] in corner and self.ubr[**'b'**] in corner and self.ubr[**'r'**] in corner:  
 self.convertmoves(**"Bi"**)  
 self.convertmoves(**"Di"**)  
 self.convertmoves(**"B"**)  
  
 elif self.dfl[**'d'**] in corner and self.dfl[**'f'**] in corner and self.dfl[**'l'**] in corner:  
 self.convertmoves(**"D"**)  
  
 elif self.dbl[**'d'**] in corner and self.dbl[**'b'**] in corner and self.dbl[**'l'**] in corner:  
 self.convertmoves(**"D2"**)  
  
 elif self.dbr[**'d'**] in corner and self.dbr[**'b'**] in corner and self.dbr[**'r'**] in corner:  
 self.convertmoves(**"Di"**)  
  
 *# corners are now in ufr or dfr positions, time to orient/move them* if self.dfr[**'r'**] == **'W'** and self.dfr[**'d'**] in corner and self.dfr[**'f'**] in corner:  
 self.convertmoves(**"D"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"Di"**)  
 self.convertmoves(**"Fi"**)  
  
 if self.dfr[**'f'**] == **'W'** and self.dfr[**'d'**] in corner and self.dfr[**'r'**] in corner:  
 self.convertmoves(**"Di"**)  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"D"**)  
 self.convertmoves(**"R"**)  
  
 if self.dfr[**'d'**] == **'W'** and self.dfr[**'f'**] in corner and self.dfr[**'r'**] in corner:  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"D2"**)  
 self.convertmoves(**"Li"**)  
 self.convertmoves(**"Fi"**)  
  
 if self.ufr[**'u'**] == **'W'** and self.ufr[**'f'**] in corner and self.ufr[**'r'**] in corner:  
 pass *# do nothing (:* if self.ufr[**'f'**] == **'W'** and self.ufr[**'u'**] in corner and self.ufr[**'r'**] in corner:  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"Di"**)  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"D"**)  
 *# now in dfr w at bottom position* self.convertmoves(**"F"**)  
 self.convertmoves(**"L"**)  
 self.convertmoves(**"D2"**)  
 self.convertmoves(**"Li"**)  
 self.convertmoves(**"Fi"**)  
  
 if self.ufr[**'r'**] == **'W'** and self.ufr[**'u'**] in corner and self.ufr[**'f'**] in corner:  
 self.convertmoves(**"Ri"**)  
 self.convertmoves(**"Di"**)  
 self.convertmoves(**"R"**)  
 self.convertmoves(**"D"**)  
 *# now in dfr w at right position* self.convertmoves(**"D"**)  
 self.convertmoves(**"F"**)  
 self.convertmoves(**"Di"**)  
 self.convertmoves(**"Fi"**)  
  
 self.convertmoves(**"Y"**)  
  
 def solveSecondLayer(self):  
 self.update\_cubies()  
 if self.u == **"W"**:  
 self.convertmoves(**"Z2"**) *# to see the edges easier, we turn the cube upside down* edges = [**"GO"**, **"OB"**, **"BR"**, **"RG"**]  
 right\_alg = [**"U"**, **"R"**, **"Ui"**, **"Ri"**, **"Ui"**, **"Fi"**, **"U"**, **"F"**]  
 left\_alg = [**"Ui"**, **"Li"**, **"U"**, **"L"**, **"U"**, **"F"**, **"Ui"**, **"Fi"**]  
  
 for edge in edges:  
 *# if edge is in middle, move it to the top* if self.fl[**'f'**] in edge and self.fl[**'l'**] in edge:  
 while **"Y"** not in self.uf.values(): *# this ensures we are not putting a piece we will need back into the second layer, this saves time* self.convertmoves(**"U"**)  
  
 for move in left\_alg:  
 self.convertmoves(move)  
  
 elif self.bl[**'b'**] in edge and self.bl[**'l'**] in edge:  
 self.convertmoves(**"Yi"**)  
 while **"Y"** not in self.uf.values(): *# this ensures we are not putting a piece we will need back into the second layer, this saves time* self.convertmoves(**"U"**)  
  
 for move in left\_alg: *# takes piece out of place* self.convertmoves(move)  
 self.convertmoves(**"Y"**) *# moves cube back to uf position* elif self.br[**'b'**] in edge and self.br[**'r'**] in edge:  
 self.convertmoves(**"Y"**)  
 while **"Y"** not in self.uf.values(): *# ensures we're not putting a piece we will need back into the second layer:saves time* self.convertmoves(**"U"**)  
 for move in right\_alg:  
 self.convertmoves(move)  
 self.convertmoves(**"Yi"**)  
 *# now edge is on top, move it to the front* if self.ub[**'u'**] in edge and self.ub[**'b'**] in edge:  
 self.convertmoves(**"U2"**)  
 elif self.ur[**'u'**] in edge and self.ur[**'r'**] in edge:  
 self.convertmoves(**"U"**)  
 elif self.ul[**'u'**] in edge and self.ul[**'l'**] in edge:  
 self.convertmoves(**"Ui"**)  
  
 *# if the front of the edge is lined up with the center, perform the algorithm to move it into the right side* if self.uf[**'f'**] == edge[0] and self.uf[**'u'**] == edge[1]:  
 for move in right\_alg:  
 self.convertmoves(move)  
 *# if the front of the edge is NOT lined up with the center,  
 # turn the cube and perform algorithm to move it into the left side* elif self.uf[**'f'**] == edge[1] and self.uf[**'u'**] == edge[0]:  
 self.convertmoves(**"Y"**)  
 self.convertmoves(**"Ui"**)  
 for move in left\_alg:  
 self.convertmoves(move)  
 self.convertmoves(**"Yi"**)  
 if self.fr[**'f'**] != edge[0]:  
 while **"Y"** not in self.uf.values(): *# ensures we're not putting a piece we will need back into the second layer:saves time* self.convertmoves(**"U"**)  
 for move in right\_alg:  
 self.convertmoves(move)  
 self.convertmoves(**"U2"**)  
 for move in right\_alg:  
 self.convertmoves(move)  
 if self.fr[**'f'**] == edge[0]:  
 pass *# goal reached* self.convertmoves(**"Y"**)  
  
 def solveYellowCross(self):  
 if self.u == **"W"**:  
 self.convertmoves(**"Z2"**)  
 self.update\_cubies()  
 cross\_completed = False  
 cross\_alg = [**"F"**, **"R"**, **"U"**, **"Ri"**, **"Ui"**, **"Fi"**]  
 while not cross\_completed:  
 *# cross completed* if ((self.ub[**'u'**] == **"Y"**) and (self.ur[**'u'**] == **"Y"**)) and ((self.uf[**'u'**] == **"Y"**) and (self.ul[**'u'**] == **"Y"**)):  
 cross\_completed = True  
  
 *# line - if horizontal line xor vertical line* elif ((self.ub[**'u'**] == **"Y"**) and (self.uf[**'u'**] == **"Y"**)) ^ ((self.ur[**'u'**] == **"Y"**) and (self.ul[**'u'**] == **"Y"**)):  
 if self.ub[**'u'**] == **"Y"** and self.uf[**'u'**] == **"Y"**: *# if horizontal line, make into vertical line* self.convertmoves(**"U"**)  
 for move in cross\_alg: *# perform move on vertical line* self.convertmoves(move)  
  
 *# dot - if other edges are not yellow* elif ((self.ub[**'u'**] != **"Y"**) and (self.ur[**'u'**] != **"Y"**)) and (  
 (self.uf[**'u'**] != **"Y"**) and (self.ul[**'u'**] != **"Y"**)): *#* for move in cross\_alg:  
 self.convertmoves(move)  
 *# L - the last possible case is the L shape* else:  
 while not (self.ub[**'u'**] == **"Y"** and self.ul[**'u'**] == **"Y"**):  
 self.convertmoves(**"U"**)  
 for move in cross\_alg:  
 self.convertmoves(move)  
  
 def matchYellowEdges(self):  
 if self.u == **"W"**:  
 self.convertmoves(**"Z2"**)  
 self.update\_cubies()  
  
 fav\_alg = [**"R"**, **"U"**, **"Ri"**, **"U"**, **"R"**, **"U2"**, **"Ri"**, **"U"**]  
 matches = []  
 d = False  
 *# while there are less than two matches, rotate until there are two matches* while len(matches) < 2:  
 if self.ub[**'b'**] == self.b:  
 matches.append(**"ub"**)  
 if self.ur[**'r'**] == self.r:  
 matches.append(**"ur"**)  
 if self.uf[**'f'**] == self.f:  
 matches.append(**"uf"**)  
 if self.ul[**'l'**] == self.l:  
 matches.append(**"ul"**)  
 if len(matches) < 2:  
 self.convertmoves(**"U"**)  
 matches = []  
  
 *# all sides are matched* if len(matches) == 4:  
 return  
 *# 2 sides are opposite* elif (**"ub"** in matches and **"uf"** in matches) or (**"ul"** in matches and **"ur"** in matches):  
 if **"ub"** not in matches:  
 self.convertmoves(**"U"**)  
 d = True  
 for move in fav\_alg:  
 self.convertmoves(move)  
 if d: *# this just corrects for the U move we did a few lines ago* self.convertmoves(**"Ui"**)  
 *# match sides again* new\_matches = []  
  
  
 *# if sides are next to each other* elif len(matches) == 2:  
 while not (self.ub[**'b'**] == self.b and self.ur[**'r'**] == self.r):  
 self.convertmoves(**"Y"**)  
 for move in fav\_alg:  
 self.convertmoves(move)  
  
 def checkdone(self): *# checks if all pieces are in the right position* match = []  
 if self.srt(self.ufr) == sorted([self.u, self.f, self.r]):  
 match.append(self.ufr)  
 if self.srt(self.ubr) == sorted([self.u, self.b, self.r]):  
 match.append(self.ubr)  
 if self.srt(self.ubl) == sorted([self.u, self.b, self.l]):  
 match.append(self.ubl)  
 if self.srt(self.ufl) == sorted([self.u, self.f, self.l]):  
 match.append(self.ufl)  
 if len(match) == 4:  
 return True  
 else:  
 return False  
  
 def positionYellowCorners(self):  
 if self.u == **"W"**:  
 self.convertmoves(**"Z2"**)  
 self.update\_cubies()  
  
 corner\_alg = [**"U"**, **"R"**, **"Ui"**, **"Li"**, **"U"**, **"Ri"**, **"Ui"**, **"L"**]  
 while not self.checkdone():  
 *# finds matched corner and performs algorithm on there* match = []  
 if self.srt(self.ufr) == sorted([self.u, self.f, self.r]):  
 match.append(self.ufr)  
 for move in corner\_alg:  
 self.convertmoves(move)  
 elif self.srt(self.ubr) == sorted([self.u, self.b, self.r]):  
 match.append(self.ubr)  
 self.convertmoves(**"Y"**)  
 for move in corner\_alg:  
 self.convertmoves(move)  
 elif self.srt(self.ubl) == sorted([self.u, self.b, self.l]):  
 match.append(self.ubl)  
 self.convertmoves(**"Y"**)  
 self.convertmoves(**"Y"**)  
 for move in corner\_alg:  
 self.convertmoves(move)  
 elif self.srt(self.ufl) == sorted([self.u, self.f, self.l]):  
 match.append(self.ufl)  
 self.convertmoves(**"Yi"**)  
 for move in corner\_alg:  
 self.convertmoves(move)  
 elif len(match) == 0:  
 for move in corner\_alg:  
 self.convertmoves(move)  
  
 def orientYellowCorners(self):  
 if self.u == **"W"**:  
 self.convertmoves(**"Z2"**)  
 self.update\_cubies()  
  
 orient\_alg = [**'Ri'**, **'Di'**, **'R'**, **'D'**]  
  
 while self.ufr[**'u'**] + self.ubr[**'u'**] + self.ufl[**'u'**] + self.ubl[**'u'**] != **"YYYY"**:  
 *# if corner isn't oriented* if self.ufr[**'u'**] != **"Y"**:  
 for move in orient\_alg:  
 self.convertmoves(move)  
  
 if self.ufr[**'u'**] == **"Y"**:  
 self.convertmoves(**"U"**)  
  
 while self.uf[**'f'**] != self.f:  
 self.convertmoves(**"U"**)  
  
 *# rotates cube so thar green is the front and white is at the top* def orientToNeutralPosition(self):  
 if self.u == **"Y"**:  
 self.convertmoves(**"Z2"**)  
 while self.f != **"G"**:  
 self.convertmoves(**"Y"**)  
  
 def solveCube(self):  
  
 self.printCube()  
 self.solveWhiteCross()  
  
 self.solveWhiteCorners()  
  
 self.solveSecondLayer()  
  
 self.solveYellowCross()  
  
 self.matchYellowEdges()  
  
 self.positionYellowCorners()  
  
 self.orientYellowCorners()  
  
 self.orientToNeutralPosition()  
  
 self.printCube()  
  
  
class CubeCV(handler):  
  
 def \_\_init\_\_(self):  
 handler.\_\_init\_\_(self)  
  
 self.width = None  
 self.height = None  
  
 self.faces = {**"red"**: 2, **"blue"**: 5, **"orange"**: 3, **"green"**: 1, **"white"**: 0, **"yellow"**: 4}  
 self.colorvalues = {**"red"**: None, **"blue"**: None, **"orange"**: None, **"green"**: None, **"white"**: None, **"yellow"**: None}  
  
 *# Reads counter file and returns the next number to use for image names* def img\_counter\_file(self):  
 counter\_file = open(**"image\_counter\_file.txt"**, **"r+"**)  
 count = int(counter\_file.read())  
  
 count += 1  
 counter\_file.truncate(0)  
 counter\_file.close()  
  
 counter\_file = open(**"image\_counter\_file.txt"**, **"w"**)  
 counter\_file.write(str(count))  
 counter\_file.close()  
  
 return count  
  
 *# Resizes webcam box size* def rescale\_frame(frame, percent=75):  
 width = int(frame.shape[1] \* percent / 100)  
 height = int(frame.shape[0] \* percent / 100)  
 dim = (width, height)  
 return cv.resize(frame, dim, interpolation=cv.INTER\_AREA)  
  
 *# Opens webcam and asks users to take picture of cube* def webcam(self):  
 global saved  
 cam = cv.VideoCapture(0)  
 cv.namedWindow(**"Cube catcher"**)  
 counter = 0  
 order = [**"red"**, **"blue"**, **"orange"**, **"green"**, **"white"**, **"yellow"**]  
 facephotos = []  
 lines = [**"Hold the cube with the WHITE side facing UP: Position the RED side in the square (space to capture)"**,  
 **"Hold the cube with the WHITE side facing UP: Position the BLUE side in the square"**,  
 **"Hold the cube with the WHITE side facing UP: Position the ORANGE side in the square"**,  
 **"Hold the cube with the WHITE side facing UP: Position the GREEN side in the square"**,  
 **"Hold the cube with the GREEN side facing DOWN: Position the WHITE side in the square"**,  
 **"Hold the cube with the GREEN side facing UP: Position the YELLOW side in the square"**]  
 img\_counter = self.img\_counter\_file()  
 on = True  
  
 try:  
 *# main loop* while on:  
 ret, img = cam.read()  
 *# gets dimensions of window* self.width = int(cam.get(3))  
 self.height = int(cam.get(4))  
  
 if not ret:  
 print(**"capture failed"**)  
 break  
 *# draws rectangle on cam* img = cv.flip(img, 1)  
 img = cv.rectangle(img, (180, 100), (460, 380), (0, 200, 0), 5) *# top left, bottom right, thickness  
 # start, end, rgb, thickness* img = cv.line(img, (273, 100), (273, 380), (200, 200, 200), 3)  
 img = cv.line(img, (367, 100), (367, 380), (200, 200, 200), 3)  
 img = cv.line(img, (180, 193), (460, 193), (200, 200, 200), 3)  
 img = cv.line(img, (180, 287), (460, 287), (200, 200, 200), 3)  
  
 img = cv.putText(img, lines[counter], (10, 30), cv.FONT\_HERSHEY\_SIMPLEX, 0.4, (00, 0, 0),  
 1) *# displays text with instructions* img = CubeCV.rescale\_frame(img, percent=150) *# makes the frame look bigger (:* cv.imshow(**"Cube catcher"**, img)  
 *# wait key waits until a key is pressed* k = cv.waitKey(1)  
 *# closes app* if cv.getWindowProperty(**"Cube catcher"**, cv.WND\_PROP\_VISIBLE) < 1:  
 on = False  
 if k % 256 == 27: *# esc key* print(**"Escape hit, closing the app"**)  
 on = False  
 *# takes picture* elif k % 256 == 32: *# space key  
 #* img\_name = **f"opencv\_**{order[counter]}{img\_counter}**.png"** facephotos.append(img\_name)  
 cv.imwrite(img\_name, img)  
 print(**"side captured"**)  
 counter += 1  
 if counter == 6: *# if we are on the last image do this  
 # print(facephotos) / remove comment to print files for test material* for file, color in zip(facephotos, order): *# puts each photo into the cube array* self.face\_filler(self.faces[color], color, file)  
  
 self.rgb\_to\_color\_cube()  
 self.display\_cube()  
 *# saved = self.cube\_to\_rotations()* on = False  
  
 cam.release()  
 cv.DestroyAllWindows()  
 except AttributeError:  
 pass  
  
 *# Turns a cube 3d list of rgb values into a cube of colors* def rgb\_to\_color\_cube(self):  
 for i, face in enumerate(self.cube):  
 for j, row in enumerate(face):  
 diffdict = {}  
 for k, facelet in enumerate(row):  
 for name, rgb in self.colorvalues.items():  
 difference = sqrt(  
 (facelet[0] - rgb[0]) \*\* 2 + (facelet[1] - rgb[1]) \*\* 2 + (facelet[2] - rgb[2]) \*\* 2)  
 diffdict[round(difference, 2)] = name  
 color = diffdict[min(diffdict)]  
 color = self.hue\_comparison(color, facelet)  
 self.cube[i][j][k] = color[0].upper()  
  
 *# this function takes what i thought to be the color of an rgb value then finds the color with the closest hue and becomes that* def hue\_comparison(self, color\_arg, rgb):  
 *# print(self.colorvalues,"#]##########")  
 # print("color and rgb", color\_arg,rgb)  
 # groups of colors which are similar* near\_groups = [(**"white"**, **"yellow"**), (**"green"**, **"blue"**), (**"red"**, **"orange"**, **"yellow"**)]  
 hue\_diffdict = {}  
 *# loops through all of the groups of similar colors* for group in (near\_groups):  
 *# if the color is in a similar group* if color\_arg in group:  
 *# find the hues of the colors in the group and compare which its closest to* for color\_name in group:  
 color\_rgb = self.colorvalues[color\_name]  
 color\_hue = self.rgb\_to\_hsv(color\_rgb)[0]  
 hue\_diffdict[abs(color\_hue - self.rgb\_to\_hsv(rgb)[0])] = color\_name  
 *# returns the name of the color with the minimum hue difference* return hue\_diffdict[min(hue\_diffdict)]  
  
  
 *# Returns color of the pixel in the coordinate position* def rgb\_to\_hsv(self, rgb): *# this function converts rgb values into hsv values* r, g, b = rgb  
 r, g, b = r / 255.0, g / 255.0, b / 255.0 *# converts rgb into a value in range 0-1* mx = max(r, g, b) *# find maximum and minimmum values* mn = min(r, g, b)  
 df = mx - mn  
 if mx == mn:  
 h = 0  
 elif mx == r:  
 h = (60 \* ((g - b) / df) + 360) % 360  
 elif mx == g:  
 h = (60 \* ((b - r) / df) + 120) % 360  
 elif mx == b:  
 h = (60 \* ((r - g) / df) + 240) % 360  
 if mx == 0:  
 s = 0  
 else:  
 s = (df / mx) \* 100  
 v = mx \* 100  
 hsv = (h, s, v)  
 return hsv  
  
 def pixel\_identifier(self, coordinate, filename=**"opencv\_frame0.png"**):  
 image = Image.open(filename)  
 rgbpixel = image.getpixel(coordinate)  
 del image  
 return rgbpixel  
  
 *# Returns average rgb value of a facelet* def getfaceletcolor(self, coordinate,  
 filename): *# input the center coord of a facelet and it will return its color in hvs* x, y = coordinate  
 x \*= 1.5  
 y \*= 1.5  
 k = 5 *# k is shift constant* rgbvals = (self.pixel\_identifier(coordinate, filename), self.pixel\_identifier((x + k, y), filename),  
 self.pixel\_identifier((x - k, y), filename), self.pixel\_identifier((x, y + k), filename),  
 self.pixel\_identifier((x, y - k), filename))  
  
 avgcol = tuple(np.mean(rgbvals, axis=0))  
 return avgcol  
  
 *# Inputs rgb values of facelet into the 3D list* def face\_filler(self, face, color, file):  
 *# this function will take center values of each facelet and run the pixel identifier function on it then get the name of the collor* x = face  
  
 self.cube[x][0][0] = self.getfaceletcolor((413, 147), filename=file)  
 self.cube[x][0][1] = self.getfaceletcolor((320, 147), filename=file)  
 self.cube[x][0][2] = self.getfaceletcolor((227, 147), filename=file)  
  
 self.cube[x][1][0] = self.getfaceletcolor((413, 240), filename=file)  
  
 self.cube[x][1][1] = self.colorvalues[color] = self.getfaceletcolor((320, 240), filename=file) *# center* self.cube[x][1][2] = self.getfaceletcolor((227, 240), filename=file)  
  
 self.cube[x][2][0] = self.getfaceletcolor((413, 333), filename=file)  
 self.cube[x][2][1] = self.getfaceletcolor((320, 333), filename=file)  
 self.cube[x][2][2] = self.getfaceletcolor((227, 333), filename=file)  
  
 *# for debugging* def test(self):  
 order = [**"red"**, **"blue"**, **"orange"**, **"green"**, **"white"**, **"yellow"**]  
 test\_sequence = [**'opencv\_red93.png'**, **'opencv\_blue93.png'**, **'opencv\_orange93.png'**, **'opencv\_green93.png'**,  
 **'opencv\_white93.png'**, **'opencv\_yellow93.png'**]  
 for file, color in zip(test\_sequence, order): *# puts each photo into the cube array* self.face\_filler(self.faces[color], color, file)  
  
 print(self.colorvalues)  
 *# print(self.cube)* self.display\_cube()  
 self.rgb\_to\_color\_cube()  
 self.display\_cube()  
 print(self.check\_legitimacy())  
  
 *# checks if inputted cube is possible* def check\_legitimacy(self):  
 facelets = []  
 for face in self.cube:  
 for row in face:  
 for facelet in row:  
 facelets.append(facelet)  
  
 for \_ in [**"W"**, **"Y"**, **"O"**, **"G"**, **"B"**, **"R"**]:  
 amount = facelets.count(color)  
 if amount != 9:  
 print(**"Color error: please edit the cube"**)  
 self.edit\_cube()  
 return True  
 else:  
 yn = input2(**"Is the cube displayed correct? (y/n)"**)  
 if yn.lower() == **"y"**:  
 return True  
 else:  
 self.edit\_cube()  
  
 def edit\_cube(self):  
 print(**"enter edit code ( 510r means, counting from zero; change face 5, row 1, facelet 0 to red"**)  
 while 1:  
  
 edit\_code = input2(**"enter edit code (xxx when complete) :"**)  
 *# if the user enters xxx, exit the function* if **"xxx"** in edit\_code.lower():  
 break  
 face, row, facelet, color = list(edit\_code[:4])  
 try:  
 self.cube[int(face)][int(row)][int(facelet)] = color.upper()  
 self.display\_cube()  
 except:  
 print(**"invalid code"**)  
  
  
*# because ursina comes with a built in function called input, the normal python input will not work  
# ths function just replaces the normal input() function*def input2(prompt=**""**):  
 print(prompt, end=**""**)  
 return sys.stdin.readline()  
  
  
class Menu(handler):  
  
 def \_\_init\_\_(self):  
 pass  
  
 def mainMenu(self):  
 global inst3D, saved, instDB  
 print(**"Menu"**)  
 print(**"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"**)  
 print(**"1: 3D Cube "**)  
 print(**"2: Input real cube "**)  
 print(**"3: Log in"**)  
 print(**"4: Sign up"**)  
 print(**"5: Score Board"**)  
 print(**"6: Exit program"**)  
  
 answer = sys.stdin.readline()[:1]  
  
 if answer == **"1"**:  
 inst3D = Cube3D()  
 instDB = CubeDB()  
 inst3D.initCube()  
  
 app.run()  
 self.menucv()  
  
 if answer == **"2"**:  
 instCV = CubeCV()  
 instCV.webcam()  
 if instCV.check\_legitimacy():  
 *# saved =* inst3D = Cube3D(cube=instCV.cube)  
 inst3D.build\_cube(instCV.cube\_to\_rotations())  
 inst3D.initCube()  
 app.run()  
 else:  
 print(**"Capture failed, try in different lighting"**)  
  
 elif answer == **"3"**:  
 instDB.log\_in()  
 elif answer == **"4"**:  
 instDB.sign\_up()  
 elif answer == **"5"**:  
 CubeDB.display\_scores()  
 elif answer == **"6"**:  
 sys.exit()  
 else:  
 print(**"Invalid input"**)  
 self.mainMenu()  
  
  
*# Run program  
##################################*run\_menu = True  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 while 1:  
 Menu().mainMenu()

Testing

Testing the 3D cube inputs and functions

<https://youtu.be/m2lyFLtp5vk>

This video showcases the mouse movement inputs, keyboard cube inputs, reset function, save function, scramble function, load function, log in function, hints function, and the scoreboard display function

Testing the Solver

<https://youtu.be/5PxbQbVWExo>

This video showcases a full algorithm solve of a scrambled Rubik’s cube.

Testing the webcam input

<https://youtu.be/gBxHhZcCylY>

This videos shows me taking images of a real Rubiks cube and transferring it to the 3D space

# First test

The Webcam cube function didn’t work in the end, the code cannot differentiate between orange, yellow and red. Here I have a picture of a face and its outputted colour values. If you compare the image to the output, you can see the red orange and yellows get mixed up quite often

A picture containing toiletry, cosmetic

Description automatically generated

['R', 'W', 'W']

['O', 'O', 'Y']

['R', 'R', 'R']

# Second test (improvements)

The amount of mixed up colours is reduced due to calculating and comparing hues. The user can now edit the wrong facelets too.

Evaluation

Here are the project objectives I decided on at the start of the project in the analysis section. I will comment on each of these in a red font for negatives or a green font for positives.

Project objectives.

## THE CUBE

* Design a data structure that holds the current state of the cube. The cube uses 2 different data structures, each data structure has it strengths and weaknesses however the weakness of one data structure is the strength of another which creates a near flawless system
* Make a function that figures out the new state of the cube after each rotation and updates the data structure. This objective was achieved by the ” handler” class. There was no pretty way to do this. Just hundreds of lines of assigning facelets to a new position for each rotation.
* Create a 3d model that displays a simulation of the cube in 3D has animations for rotations. This objective was achieved by the Cube3D class
* Allow user to drag mouse around screen to make the cube rotate around its axis’. Objective achieved, the controls are very simple and intuitive.
* Map the path of the mouse movements when the user drags the mouse on the cube to figure out which face they are trying to rotate. This objective was difficult as the program must know which side is the front face, this was achieved by the Euclidean distance formula. In order to rotate the top face, you also need to know the rotation of the cube. The Ursina function “camera.world\_rotation” gives the rotation of the camera however the output is unreliable.
* Allow user to use keyboard to rotate faces (D = Right, A = Left, W = Up, S = Down, Q = Front, E = Back + and any of these letters with shift for the anticlockwise rotation of the face). since the mouse controls were so intuitive, this was not needed in any way.
* Allow users to undo a move by storing previous moves in a stack data structure and popping from the stack to undo a move.

## THE SOLVER SCRIPT

* Write code finds the solution to the Rubik’s cube when given the variable of the state of the cube. The code will solve the using the Beginners Method, the same one a human would use, and return cube notation for the solve. This object was achieved in the Solver class
* Make the solver script able to give hints of what set of moves the user must use next based on it analysing what stage of completion the cube is on (e.g. if the white cross is formed, the program will see than and respond by giving the hint “move the white corners to complete the white face using the U’ R’ U R, or U L U’ L’ algorithms). The program can identify which state the cube is currently on. There is a hints button for the user to click.

## THE HINTS

* Create good visual hints. Text based hints deliver more detail for a newbie.
* Design animations to show the next steps needed. If the user holds z, the cube will complete its self, this is better than an animation.

## COMPUTER VISION

* Write code to allow webcam to locate the cube. Rather than using a complicated tracking system, I went with a simpler more reliable stationary grid system which works well.
* Design program to let webcam identify and store the colours on the cube and their positions into a variable. The colours are not identified correctly 100% of the time. To fix this, the user can easily edit the cube
* Prepare prompts and instructions telling the user to rotate the cube to show all 6 faces in the right orientation. Instructions are printed clearly on the screen

## The Robot

I did not have the tools to physically construct a wooden frame for the robot. Therefore I scrapped this objective as it wasn’t important in teaching a beginner how to solve the cube.

* Construct robot frame and wire the electronic components.
* Translate cube notation into a set of moves for the robot to execute.
* Write 20 move random scramble algorithm for robot to follow.
* Create potentiometer inputs to slow down the speed of the robot.

## THE TIMER

* Design a relational database to save the dates and speeds of solves of each user. This objective is completed in the CubeDB class of the rubiks\_database.py file. The database also stores saves of the cubes
* Create graphical outputs of speed to show the user their progression.

I will add this in the future.

# With more time I would like to:

* Build the solving robot
* Create graph output to show progress
* Improve hints so that a complete beginner can learn just from the program rather than just practice
* Make the database networked so there is a global scoreboard

# Feedback