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**

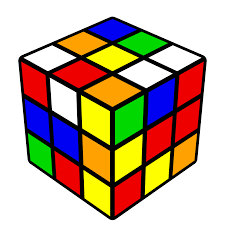
# rwari

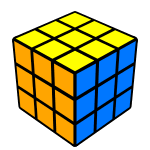
Analysis

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.

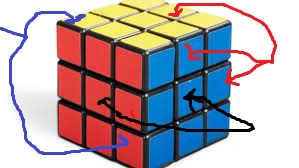
A timer that will log how quick your solves are, save the speeds and dates into a relational database which stores users and their speeds, and plot graphs with the data in the tables.

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

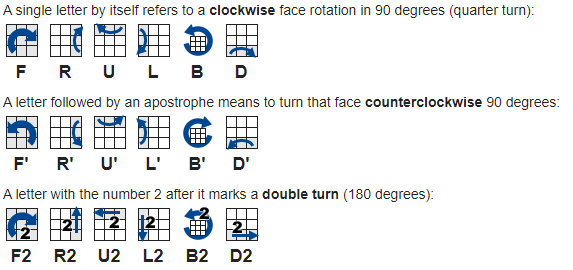
Edge pieces

Centre pieces

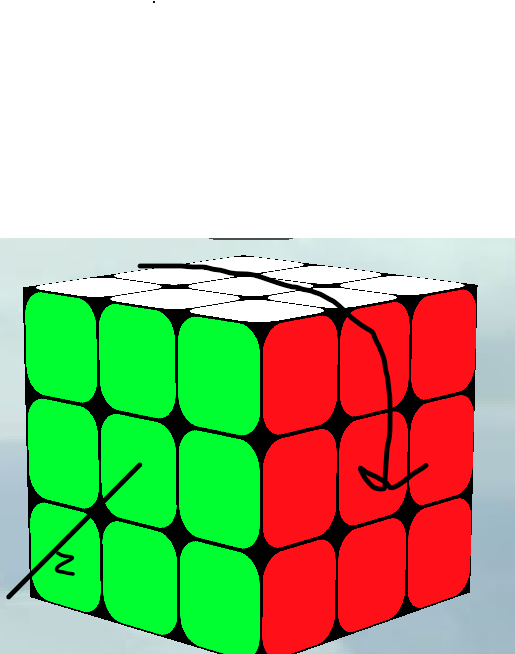
# Cube notation

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**

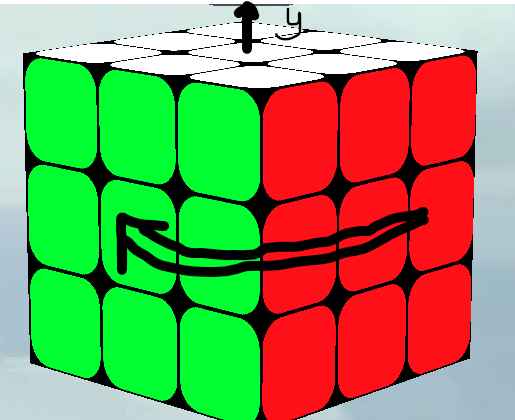
For the Beginners method, the method which the code will follow, you must know the **F** (front), **B** (back), **R** (right), **L** (left), **D** (down), **U** (up) and their inverses, e.g. R’ (pronounced as Right prime) AKA Ri (pronounced Right inverse), as explained below.



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

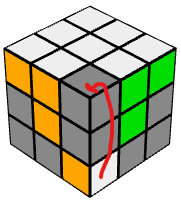
## Solving Algorithms/Methods

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.

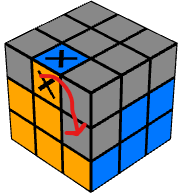
# 1 The Beginners method

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.

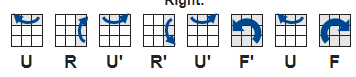
1. White cross - Create a white cross pattern on the top by lining the white edge piece with the centre pieces.



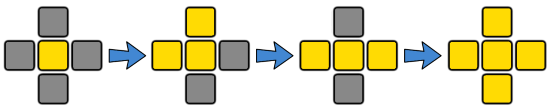
1. White corners – use the Ri, Di, R formula to move the white corners into a place where both each of the sides matches the centre colour of the face it is on.



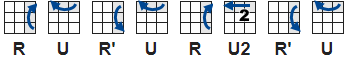
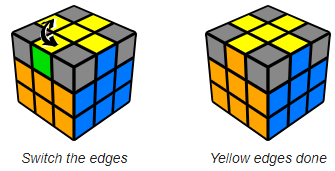
1. Middle layer – uses the U, R, Ui, Ri, Ui, Fi, U, F formula to move an edge piece to the right and its inverse Ui, Li, U, L, U, F, Ui, F to move the edge pieces to the left, which solves the solve middle layer.



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

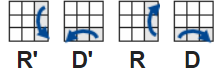
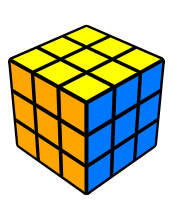
1. Position yellow corners – to solve the last layer corners, first we have to get them to the right spot, orientation doesn’t matter in this step. Find a piece which is already in the right place (if there are non in place, perform the algorithm on any side), move it to the right-front-top corner then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

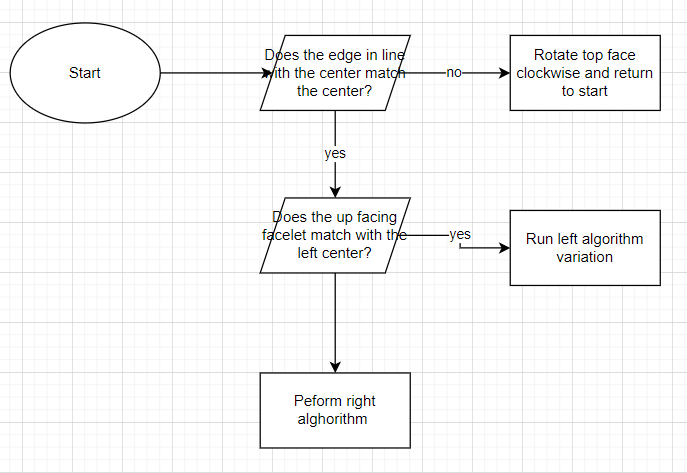
## THE TIMER

* 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.

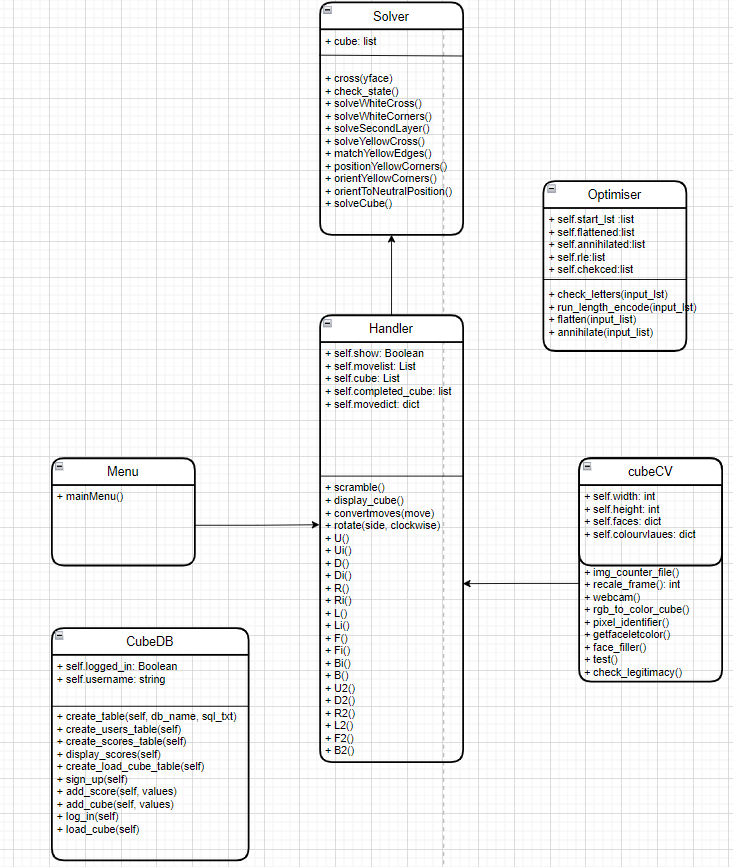
For the computer vision part of the code, I used OpenCV

# 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:

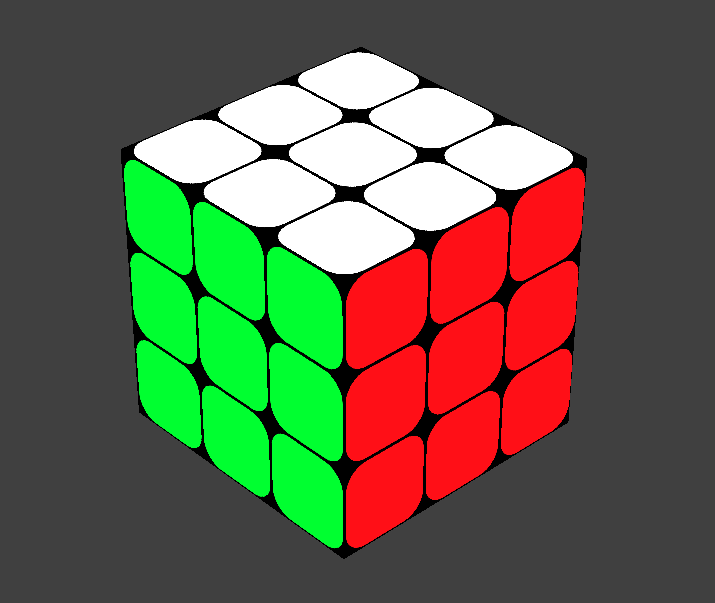


rotate\_cube\_Di()

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

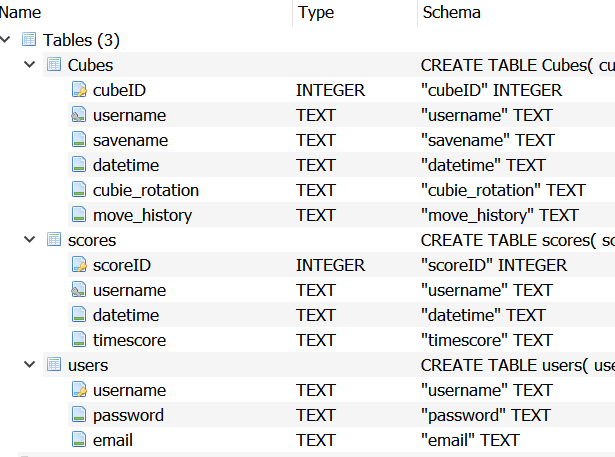
uf == {'u': W, 'd': '', 'f': f, 'b': '', 'r': '', 'l': ''}:

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)

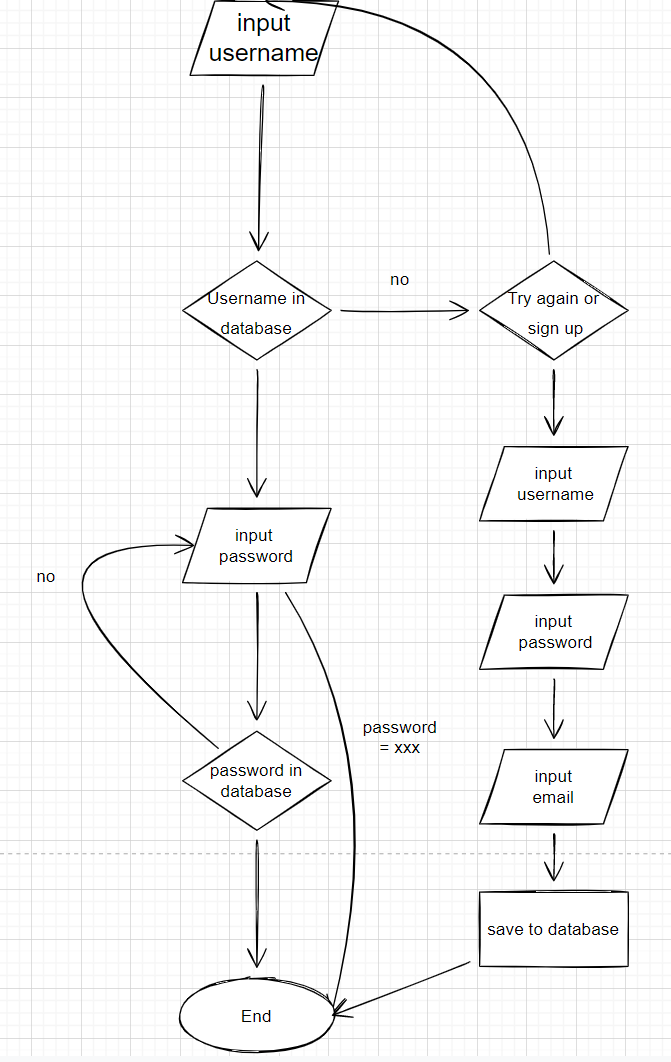


Saving cubes

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

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