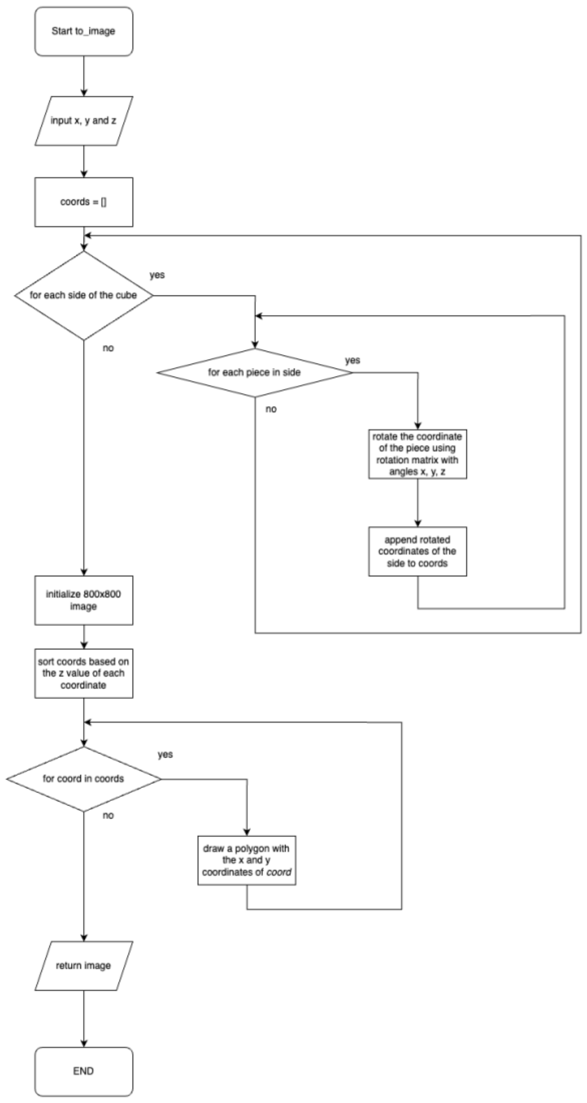
Criterion C: Development

1. Generate the 3D image of the cube.


Start to_image
input x, y and z
coords = []
for each side of the cube
yes
no
for each piece in side
yes
no
rotate the coordinate of the piece using rotation matrix with angles x, y, z
append rotated coordinates of the side to coords
initialize 800x800 image
sort coords based on the z value of each coordinate
for coord in coords
yes
no
draw a polygon with the x and y coordinates of coord
return image
END

Techniques used:

- Nested For loops for 3D list traversal
- Matrix multiplication
- Parallel lists
- Sorting

I had to initiate parallel lists to assign each side of the cube a starting coordinate and expand direction (0 represents the x-axis, 1 -> y-axis, 2 -> z-axis). As shown below, the first face starts at the coordinate (-1, 1, -1) and would expand in the z and x-axis.

```
start = [[-1, 1, -1], [-1, 1, -1], [-1, 1, 1], [1,
1, 1], [1, 1, -1], [-1, -1, 1]]
direction = [(2, 0), (1, 2), (1, 0), (1, 2), (1,
0), (2, 0)]
```

A triple nested for-loop is used to iterate through all the pieces of the cube and drawing them at the coordinate given by the parallel lists. By drawing each piece separately instead of the entire face at once, I can add gaps between each piece. This is useful because it lets the user see the pieces at the back and bottom face, making look-ahead easier for the user during a virtual solve (Figure 1).

The 3D coordinate of each piece is transformed to 2D by multiplying it by a rotation matrix then taking the x and y coordinates of the product. The z coordinate of the product is also stored and then sorted so that the pieces are drawn starting with pieces at the back to pieces at the front. Otherwise, the output could turn out like Figure 2, which drew the blue face above green face, making the color scheme wrong (blue is supposed to be on the right of red when white is top). Having the color scheme correct is important because it also makes look-ahead easier for the user. The rotation matrix is given by

```
rotation_matrix = [
    [cos(y)*cos(z), sin(x)*sin(y)*cos(z)-cos(x)*sin(z),
cos(x)*sin(y)*cos(z)+sin(x)*sin(z)],
    [cos(y) * sin(z), sin(x) * sin(y) * sin(z) + cos(x) * cos(z), cos(x) *
sin(y) * sin(z) - sin(x) * cos(z)],
    [-sin(y), sin(x)*cos(y), cos(x)*cos(y)]
]
```

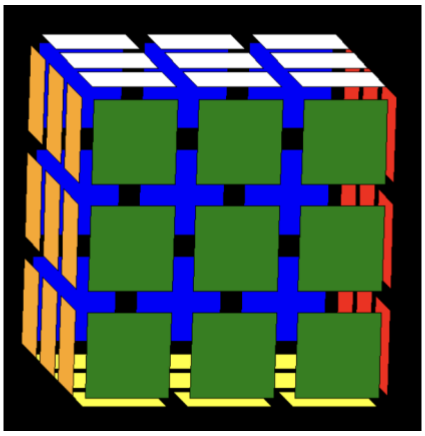


Figure 1: Working example of to_3d

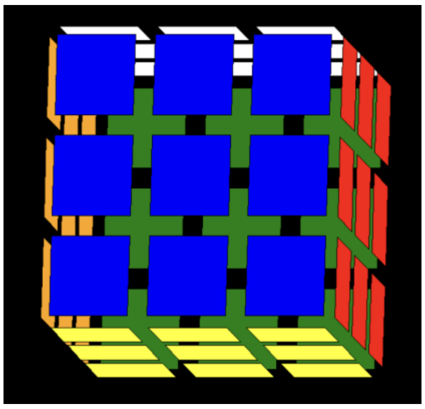
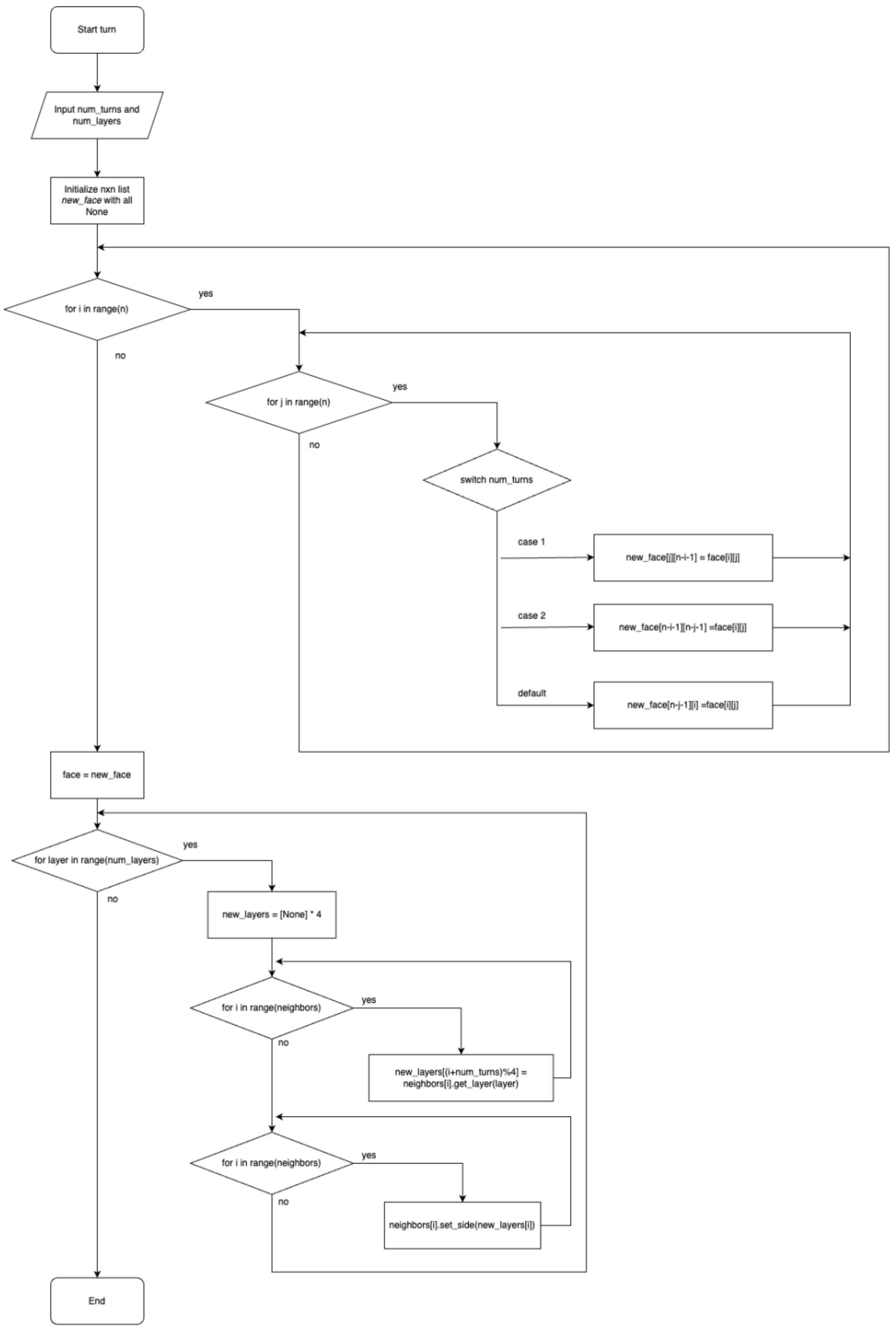


Figure 2: to_3d without sorting z

2. Turn a face


Start turn
Input num_turns and num_layers
Initialize nxn list new_face with all None
for i in range(n)
yes
no
for j in range(n)
yes
no
switch num_turns
case 1
new_face[j][n-i-1] = face[i][j]
case 2
new_face[n-i-1][n-j-1] =face[i][j]
default
new_face[n-j-1][i] =face[i][j]
face = new_face
for layer in range(num_layers)
yes
no
new_layers = [None] * 4
for i in range(neighbors)
yes
no
new_layers[(i+num_turns)%4] = neighbors[i].get_layer(layer)
for i in range(neighbors)
yes
no
neighbors[i].set_side(new_layers[i])
End

Techniques used

- For-loops for 2D list traversal
- Rotating 2D list

The algorithm loops through each piece of the face and it would be placed into a new nxn list corresponding to where the piece would end up after the turn. The pieces are not directly put into the original list because it would overwrite pieces that would need to be rotated later in the loop. For example when turning a 7x7 face clockwise, if I directly put the piece in face[2][5] to face[5][4], I would not know what the original face[5][4] is by the time I iterate to it (see Figure 3-Figure 5).

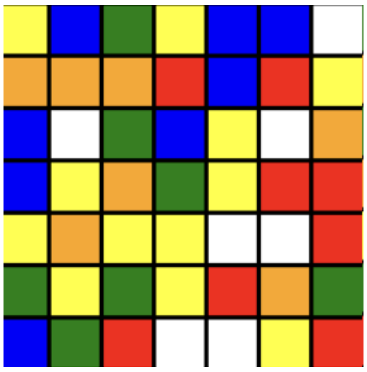


Figure 3: Original Face

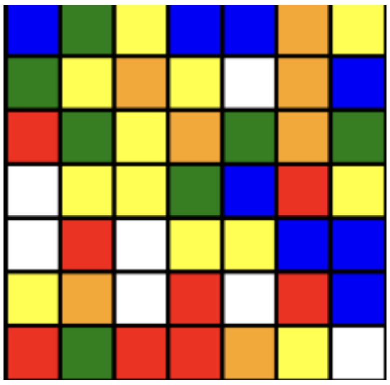


Figure 4: Face turned clockwise correctly

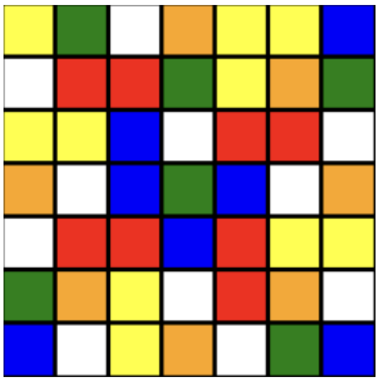


Figure 5: Face turned clockwise incorrectly

When turning a face, the layer attached to the face is also rotated. For example, when turning the white face, not only the white face is turned, but the layer right below it is also rotated (see Figure 6). These rotations are done by saving the rotated states to a new list, similar to rotating the face. Then, I set the faces and layers to the new lists.

I chose to store an nxn cube as 6 *nxn* lists of integers (0=white, 1=orange, 2=green, 3=red, 4=blue, 5=yellow) because it allows me to easily alter each face.

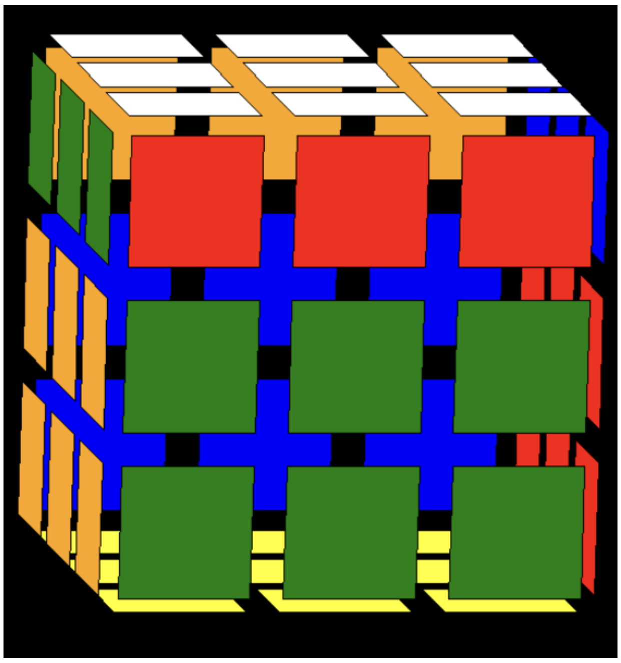
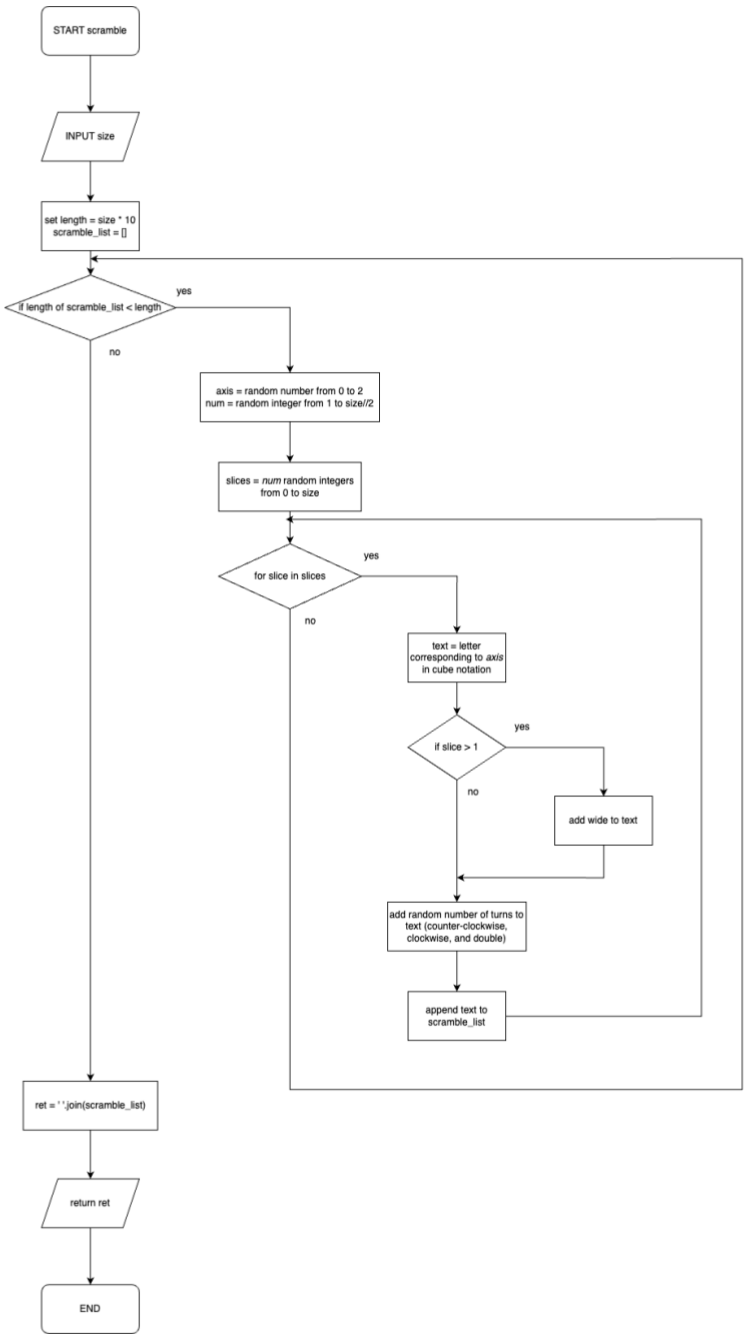


Figure 6: Example U turn

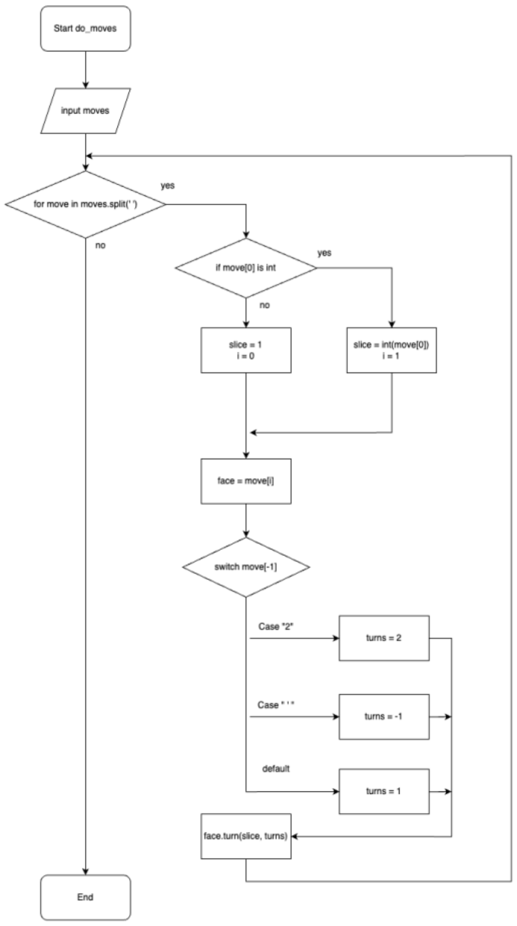
3. Generate scramble


START scramble
INPUT size
set length = size * 10
scramble_list = []
if length of scramble_list < length
yes
no
axis = random number from 0 to 2
num = random integer from 1 to size//2
slices = *num* random integers from 0 to size
for slice in slices
yes
no
text = letter corresponding to *axis* in cube notation
if slice > 1
yes
no
add wide to text
add random number of turns to text (counter-clockwise, clockwise, and double)
append text to scramble_list
ret = ' '.join(scramble_list)
return ret
END

The length of the scramble varies for each size of cube. Since a larger cube has more pieces, longer scrambles for larger cubes are needed to ensure that the cube is thoroughly scrambled. For each iteration of the while loop, the current axis must be different from the previous; otherwise, a new move may cancel a previous move. For example, if an R move is followed by a R', it's the same as not moving at all. Similarly, I chose to use random.sample over random.choices to avoid picking a slice more than once, or else redundant moves may occur.

```
while len(scramble_list) < length:
    axis = prev
    while axis == prev:
        axis = random.randint(0, 2)
    prev = axis
    num = random.randint(1, size // 2)
    slices = random.sample(range((size - 1) if size % 2 else size),
k=num)
), k=num)
```

4. Reading Cube Notation



The moves list is first split by a space as each move could be considered independently. Since cube notations have many different rules, a series of if-else statements and switch statements were used (Python does not have switch statements so the actual code would be written with if-elif-else statements). In addition, there should also be error-checking in the code, so if the user input an invalid sequence of moves, the bot should tell the user that the move is invalid.

5. OOP

Instead of storing time as a float, I store the times as an instance of the Time class because the time needs to include special states, such as +2 and DNF. This also allows me to attach additional information like the scramble and event name of the solve.

Furthermore, various inheritance was used to reduce redundant code. Since most views (buttons, drop down menu, etc) have to store and alter information of a solve, I created a subclass of nextcord.ui.View that contains information of a solve, and all the views would inherit from this subclass.

6. Top-Down Approach

For this product, I decided to use a top-down approach because a draft of the commands needed is already made. By having the main structure completed (coding the commands), it makes coding the remaining components like the cube and timer much more organized.

7. Bibliography

Admin (2022) *Matrix multiplication: How to multiply matrices: Formula & examples, BYJUS*. BYJU'S. Available at: https://byjus.com/maths/matrix-multiplication/#:~:text=Matrix%20multiplication%2C%20also%20known%20as,X%20%3D%20AB.

C.B. Fox Last Modified Date: January 30 (2023) *What is a 3D projection?*, *Easy Tech Junkie*. Available at: https://www.easytechjunkie.com/what-is-a-3d-projection.htm.

Difference between Bottom-Up Model and Top-Down Model (no date) *Code studio*. Available at: https://www.codingninjas.com/codestudio/library/top-down-vs-bottom-up-programming-whats-the-difference.

Inplace rotate square matrix by 90 degrees: Set 1 (2023) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/inplace-rotate-square-matrix-by-90-degrees/.

Rotation matrix (no date) *from Wolfram MathWorld*. Available at: https://mathworld.wolfram.com/RotationMatrix.html.