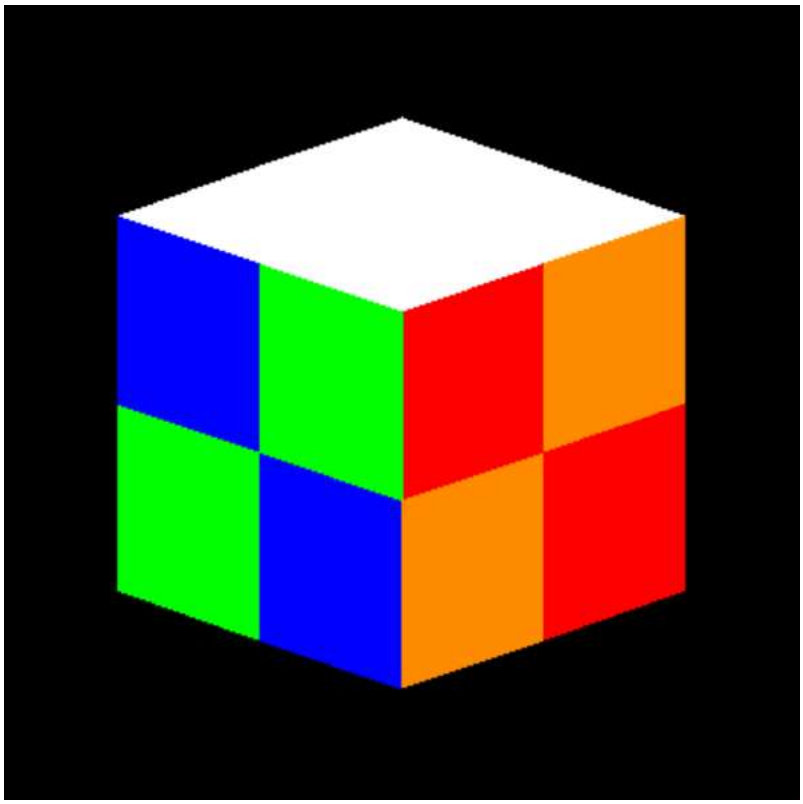
2x2-Zauberwürfel in Pygame zeichnen, steuern und lösen



Janis Wäspi

Gymnasium Biel-Seeland

Klasse 24f – Maturajahrgang 2024

Betreuung: Lukas Schaffner

Abstract

In meiner Maturaarbeit möchte ich herausfinden, ob und wie ich es schaffen kann, einen dreidimensionalen 2x2-Zauberwürfel zu programmieren, der sich selbst auf verschieden effiziente Arten lösen kann. Dafür verwende ich die Programmiersprache Python und die Bibliotheken Pygame, Numpy und Py222. Nachdem ich verschiedene Lösungsalgorithmen programmiert hatte, erstellte ich Statistiken zu ihren Effizienzen. Daraus werde ich ein Fazit ziehen. Ich untersuchte die Ergebnisse sowohl auf Drehungs- als auch Zeiteffizienz und die Kombination der beiden. Ich zog auch die Extremwerte in Betracht. Die Analyse ergab, dass ein Zufallsalgorithmus nicht effizient ist, ein optimierter Zufallsalgorithmus effizienter ist, und ein perfekter Algorithmus auch perfekte Ergebnisse liefert. Ich wusste zu Beginn nicht genau, in welchem Bereich die Werte zu erwarten sind, deshalb waren die Werte auch für mich eine Überraschung. Es gab sowohl positive als auch negative Überraschungen.

Somit kann ich behaupten, dass ich einen selbstlösenden 2x2-Zauberwürfel mit verschiedenen Lösungsalgorithmen erfolgreich programmiert habe. Es gab einige logische und technische Herausforderungen, die ich meistern musste. Abschliessend bin ich jedoch sehr zufrieden mit dem Endprodukt. Die Auswertung der Ergebnisse stimmt mit meinen Erwartungen überein und rechtfertigt den Zusatzaufwand, um das Programm zu optimieren.

Inhaltsverzeichnis

1. Vorwort

Zauberwürfel begeistern mich seit Februar 2020. Seither findet man mich immer wieder mit einem Zauberwürfel in der Hand und meine Zauberwürfelsammlung wächst immer weiter. So fand auch ein 2x2-Zauberwürfel den Weg in meine Sammlung. Diesen 2x2-Zauberwürfel kann ich in weniger als 10 Sekunden lösen, den 3x3-Zauberwürfel in weniger als 20 Sekunden. Doch kann ich dieses Wissen auch in ein Computerprogramm einbauen, welches den Würfel zeichnen, steuern und lösen kann? Genau dieser Frage werde ich in meiner Maturaarbeit nachgehen. Zuerst muss ich jedoch einen 2x2-Zauberwürfel in einer digitalen Form vor mir haben. Dies umzusetzen war der erste Schritt meiner Arbeit. Der zweite Schritt bestand darin, den Würfel so zu programmieren, dass er sich selbst löst. Sobald ich dies geschafft habe, werde ich eine Analyse durchführen. Damit diese Analyse aussagekräftig ist, brauche ich viele Daten. Diese Daten werde ich analysieren, interpretieren und daraus Rückschlüsse auf mein Programm ziehen. Diese Rückschlüsse werden einen Einblick in die Effizienz meiner Algorithmen geben.

Im Verlaufe meiner Arbeit bekam ich immer wieder Unterstützung. Als Erstes bedanke ich mich bei meinem Betreuer Lukas Schaffner, der mir immer wieder hilfreiche Ratschläge in mir unbekannten technischen Gebieten gab, mir Programme vorschlug und Ressourcen zur Verfügung stellte. Ich danke auch Daniela, Tanja und meiner Mutter für die wertvollen Korrekturen und Verbesserungsvorschläge, und Samuel für seine Tipps. Zum Schluss möchte ich auch meiner Familie und Freunden danken, die mich unterstützten und spannende Fragen stellten, die mich anregten.

2. Einleitung

2.1. Ziel

Ich werde versuchen, einen 2x2-Zauberwürfel in Python Pygame zu programmieren, um mit diesem auf praktische Art und Weise interagieren zu können und ihn lösen zu können. Lösen kann man ihn entweder manuell oder mit unterschiedlich effizienten Algorithmen. Diese Algorithmen versuche ich zu programmieren und zu verbessern. Ich werde zu einem funktionierenden Algorithmus aus gesammelten Daten Statistiken erstellen, diese auswerten, interpretieren und daraus Schlüsse ziehen. Diesen fertigen Code werde ich auf GitHub[1] allen zur Verfügung stellen.

2.2. Methode

Der Hauptteil meiner Arbeit besteht aus Programmieren. Ich musste verschiedene neue Elemente kennenlernen und erlernen. Dies gestaltete sich zum Teil einfacher oder schwieriger. Zuerst war mein Ziel, einen Lösungsalgorithmus zu programmieren, der den Würfel gleich löst wie ich. Da sich jedoch das Erstellen des 3D-Modells schwieriger gestaltete als erwartet, wurden meine Algorithmen simpler und realistischer umzusetzen. Somit ist meine Methode *ein Computerprogramm erstellen.*

[1] https://github.com/No0ne155/MA_2x2Cube

3. Theorie

3.1. Zauberwürfel

Der 3x3-Zauberwürfel wurde 1974 von Erno Rubik erfunden. Erno Rubik wurde am 13. Juni 1944 in Budapest geboren. Er ist Bildhauer und Professor in Architektur. Um seine Architekturstudenten räumliches Denken lehren zu können, entwickelte er den Zauberwürfel (Werneck, 1981). Ebenfalls im Jahre 1980 wurde der 3x3-Zauberwürfel mit dem Sonderpreis *Das beste Solitärspiel* (o.V., Spiel des Jahres, 2000) ausgezeichnet. Dieses beliebte 3D-Puzzlespiel bekam 1980 von Erno Rubik einen kleinen Bruder, den 2x2-Zauberwürfel.

Der 2x2-Zauberwürfel, auch Pocket Cube genannt, ist um einiges einfacher zu lösen als der 3x3-Zauberwürfel. Dies liegt daran, dass der 2x2-Zauberwürfel gleich wie ein 3x3-Zauberwürfel gelöst werden kann. Einziger Unterschied ist, dass 4 Schritte, welche die beim 2x2-Cube[2] inexistenten Kantenteile sortieren, weggelassen werden können. Es sind also 4 Algorithmen weniger erforderlich, die man kennen muss. Durch die fehlenden Kanten- und Mittelsteine hat der 2x2-Zauberwürfel mit 3'674'160 möglichen Kombinationen sage und schreibe 43’252’003’274’486’181'840 mögliche Kombinationen weniger als der 3x3-Zauberwürfel.

Der 3x3-Zauberwürfel hat 43’252’003’274’489’856'000 durch Drehen erreichbare Kombinationen. Um diese Zahl zu visualisieren, nehmen wir einmal an, dass wir jede Sekunde eine neue Kombination erreichen. Es würde 1'371'512'026'715 Jahre dauern, bis man jede mögliche Kombination einmal gefunden hätte. Dies ist in etwa so lange wie 994-mal vom Urknall vor 13,8 Mrd. Jahren bis heute!

Der 2x2-Zauberwürfel hat nur 3'674'160 mögliche Kombinationen. Diese Zahl berechnet man wie folgt:

$$\frac{8! \times 3^7}{24} = 3’674’160$$

Das erste Eckteil hat acht mögliche Positionen, die es einnehmen kann. Das zweite nur noch sieben. Mit 8! kann man somit berechnen, wie viele Möglichkeiten es gibt, die 8 Ecksteine am Cube zu verteilen. Zusätzlich kann das Eckteil in dieser Position noch 3 verschiedene

[2] In meiner Arbeit verwende ich die Wörter Zauberwürfel, Cube und Würfel ohne unterschiedliche Bedeutung.

Orientierungen annehmen. Dies wird mit 3^7 ausgedrückt. Es ist nicht 3^8, da die Orientierung des letzten Eckteils durch die Orientierung der anderen 7 Eckstücke bereits gegeben ist. Hätte die letzte Ecke eine andere Orientierung, wäre der Cube unlösbar. Zum Schluss kann man einen Cube noch auf 24 unterschiedliche Arten halten, die aber alle die gleiche Farbkombination haben, weshalb es noch durch 24 dividiert wird. Gerechnet mit einer neuen Kombination pro Sekunde, würde es nur 42 Tage dauern, bis alle 3'674'160 möglichen Kombinationen erreicht wären. Ich habe die beschriebene Formel selbst hergeleitet. Sie lässt sich aber auch von Wikipedia[3] bestätigen.

Um zu betiteln, welche Drehung man gemacht hat, verwendet man eine spezielle Schreibweise. Die World-Cube-Association[4] verwendet diese ebenfalls. Es werden die englischen Anfangsbuchstaben einer Seite verwendet. Beim 2x2-Zauberwürfel spielt die Gross-/Kleinschreibung keine Rolle, bei grösseren Cubes schon. Die Buchstaben sind:

Buchstabe	Englisch	Deutsch	Buchstabe	Englisch	Deutsch
U	Up	Oben	R	Right	Rechts
D	Down	Unten	F	Front	Vorne
L	Left	Links	B	Back	Hinten

Tabelle 1: WCA-Notation

Diese Namensgebung verwendete ich auch in meinen Programmen, da sie in der Welt des Zauberwürfels Standard ist. Es gäbe auch eine deutsche Version davon, die mir aber nicht geläufig ist.

[3] https://en.wikipedia.org/wiki/Pocket_Cube#Permutations
[4] https://www.worldcubeassociation.org/regulations/#article-12-notation

3.2. Python

Für die Programmierarbeit verwende ich die Programmiersprache Python. Python ist laut Erfinder Guido van Rossum eine „Programmiersprache auf hoher Ebene mit der Lesbarkeit von Code als zentrale Entwurfsphilosophie und einer Syntax, die Programmierern das Ausdrücken von Konzepten in wenigen Codezeilen ermöglicht“ (mattwojo, 2023). Dies bedeutet, dass Python einfach zu lesen, verstehen und zu schreiben ist, und in wenigen Zeilen viel erstellt werden kann. Mit Python kann fast alles programmiert werden.

Im Laufe meiner Arbeit werden immer wieder Code-Ausschnitte vorkommen. Diese sind entweder in `spezieller Formatierung` oder mit schwarzem Hintergrund und Farbcodierung wie in VSCode, welches ich zum Programmieren verwendete, dargestellt. Nun erkläre ich noch einige Konzepte, Strukturen und Begriffe, die im Laufe meiner Arbeit immer wieder vorkommen und eine zentrale Rolle spielen.

3.2.1. Variable

Damit man in Python einen Wert speichern kann, gibt es mehrere Möglichkeiten. In meiner Arbeit verwende ich 3 verschiedene Arten. Ich verwende Variablen, Listen und Numpy-Arrays. Bei Namensgebung von Variablen in Python spielt die Gross-/Kleinschreibung eine grosse Rolle.

```
x = 5
```

In diesem Beispiel ist `x` die Variable, und `5` der darin gespeicherte Wert. Variablen kann man im Gegensatz zu Konstanten verändern. Der gespeicherte Wert kann verschiedene Inhalte haben. So kann man in einer Variablen Ganzzahlwerte, Kommazahlwerte, Buchstaben und Wahrheitswerte abspeichern.

3.2.2. Liste

Eine Liste ist eine Sammlung aus Datenwerten, die in einem Namen abgespeichert werden. Die Werte innerhalb einer Liste können verschiedene Datentypen aufweisen. Werte innerhalb einer Liste werden durch Kommas getrennt. Innerhalb einer Liste kann auch eine zweite Liste erstellt werden.

```
zahlen = [4, 5, [7, 8]]
```

Um einen Wert aus einer Liste abzufragen, verwendet man die Eingabe `listennamen[element]`. Beispielsweise würde `zahlen[0]` den Wert `4` ausgeben, da das erste Element der Liste abgefragt wird. Die Zählung der Elemente beginnt bei null. Das dritte Element von `zahl` ist eine weitere Liste. Solche verschachtelten Listen verwende ich ziemlich oft. Um einen Wert aus einer verschachtelten Liste abzurufen, verwendet man eine verschachtelte Abfrage. `zahlen[2][1]` beinhaltet den Wert `8`. Ich rufe die Liste an der dritten Position auf, und dort das zweite Element.

3.2.3. Funktionen

Python hat viele praktische Funktionen bereits eingebaut. `print('text')` z. B. druckt `text` in die Konsole, man kann jedoch auch selbst noch weitere Funktionen schreiben.

Um eine eigene Funktion zu erstellen, verwendet man die Schreibweise `def Funktionsname(parameter)`. Darunter gibt man ein, was die Funktion machen soll. So könnte man z. B. eine Funktion erstellen, die als Parameter zwei Zahlen nimmt und diese danach miteinander multipliziert. So lassen sich repetitive Prozesse vereinfachen.

3.2.4. Klasse

Eine Klasse ist ein objektorientierter Strukturtyp in Python, der zahlreiche Möglichkeiten bietet, die man einfach nutzen kann. Klassen ermöglichen es, mehrere Objekte gleicher Bauweise zu erstellen und von diesen Objekten Daten abzurufen und Befehle auszuführen. Ich verwende Klassen, damit ich die 8 Eckteile des Cube identisch erstellen und bewegen kann.

Beim Erstellen eines Objektes kann man mehrere Parameter mit dem selbst gewählten Namen mitgeben, die in der Klasse gespeichert werden. Baut man nun selbst noch Funktionen in die Klasse ein, kann man diese Funktion durch den Befehl `objektname.funktionsname()` aufrufen. Dies ermöglicht es, dass das Objekt z. B. durch die Funktion bearbeitet wird.

3.2.5. Pygame

Pygame[5] ist eine Python-Bibliothek, die auf einfache Art und Weise ermöglicht, visuelle Spiele in Python zu programmieren. Mein Spiel ist der Zauberwürfel. Mit Pygame kann ich ein Fenster öffnen, in welchem ich mit verschiedenen Pygame-Befehlen Formen zeichnen kann. Pygame

[5] https://www.pygame.org/wiki/GettingStarted

kann ebenfalls Tasten- und Mausabfragen durchführen, was für mein Projekt sehr praktisch ist.

3.2.6. Numpy

Numpy[6] ist eine weitere Python-Bibliothek, welche viele mathematische Funktionen enthält. Der Fokus dieser Bibliothek liegt auf mehrdimensionalen Arrays und Matrizen. In meinem Projekt verwende ich Numpy-Arrays, um Vektoren abzuspeichern und mit ihnen Berechnungen durchzuführen. Numpy wird normalerweise beim Importieren mit `np` abgekürzt, damit man nicht für jede Funktion *Numpy* ausschreiben muss.

Numpy-Arrays sind grundlegend Vektoren in Python. Vektoren könnten auch in einer Liste abgespeichert werden, jedoch kommt es bei der Verwendung von Listen schnell zu einem Referenzfehler, welcher schwer zu finden ist. Numpy-Arrays sind vom Aufbau her sehr ähnlich wie normale Python Listen, haben aber für meine Arbeit praktischere Verwendungsmöglichkeiten.

[6] https://numpy.org/install/

4. Projekt

Das Ziel meines Projektes ist, einen 3-dimensionalen 2x2-Rubik's Cube in Python Pygame anzuzeigen, an dem ich selbst Drehungen vornehmen kann. Zusätzlich soll das Programm ihn selbst lösen können. Im Idealfall möchte ich mehrere Lösungsalgorithmen implementieren, die unterschiedlich effizient sind. Um dies zu erreichen, war mir von Beginn an klar, dass ich dazu eine Klassenstruktur verwende. Ich arbeitete im Freifach Gamedesign bereits mit Klassen in C++, schrieb jedoch noch nie zuvor eine eigene Klasse. Ebenfalls war mir klar, dass ich die Grafikbibliothek «Pygame» dazu verwenden werde. Ohne Vorkenntnisse legte ich einfach los und probierte diese für mich neuen Bereiche von Python aus. Für das Projekt verwendete ich zwei Python-Bibliotheken, die zuerst noch installiert werden mussten. Diese heissen Numpy und Pygame.

4.1. Tutorial

Um Pygame zu erlernen, folgte ich dem Tutorial *Pygame: A Primer on Game Programming in Python* (Fincher, 2015) und erstellte damit mein erstes funktionierendes Pygame-Programm. Ich lernte, wie ich Pygame installiere, initialisiere, Objekte anzeige und mit diesen Objekten interagieren kann. Weiter erstellt das Tutorial mithilfe einer Klasse einen Spieler, der sich auf Tastenbefehl bewegt. Ich las im Tutorial noch zu weiteren Themen wie z. B. Audio und Bilder. Da ich jedoch nicht sah, wie diese Bereiche beim Anzeigen eines Zauberwürfels helfen sollten, hörte ich auf zu lesen. Stattdessen begann ich mein eigenes Programm zu schreiben.

4.2. Version 1

Für die erste eigene Version kopierte ich das Tutorial, das ich bisher hatte, und fügte es in eine neue Python-Datei ein. Ich zeichnete manuell, wie im Tutorial gelernt, die erste Ecke des 2x2-Cube ein. Alle weiteren Ecken zeichnete ich ebenso ein, bis eine 2D-Repräsentation eines 2x2-Zauberwürfels auf dem Bildschirm erschien. Als nächstes erstellte ich eine Klasse, genannt `Cube`, welche die Parameter `pos` für Position und `colors` für Farben entgegennahm. `pos` nimmt eine 2-dimensionale Liste mit 3 Listen, die jeweils 2 Elemente enthalten, entgegen. Für die erste Ecke sieht `pos` wie folgt aus: `[[150, 90],[400,190],[50, 190]]`. `colors` ist ebenfalls eine Liste, die jedoch nur 3 Elemente, nämlich eines pro Farbquadrat, enthält.

Die einzelnen Listen in `pos` stehen für jeweils eine Kachel. Die Farbe der Kachel wird von der Zahl in `colors` bestimmt. Innerhalb einer der drei Listen in `pos` steht das erste Element für die Koordinate auf der x-Achse und das zweite Element für die Koordinate auf der y-Achse. So wird der Startpunkt des Quadrats festgelegt. Die Grösse aller Kacheln ist identisch, weshalb ich sie nicht für jede Kachel einzeln angeben muss. Das Initialisieren eines solchen Kachel-Trios sieht wie folgt aus:

```
cubicle0 = Cube([[150, 90],[400,190],[50, 190]],[0,4,1])
```

Diese Codezeile erstellt die weiss-blau-orange Ecke. In diesem Programm bezeichnete ich die acht Ecken `cubicle` mit der korrespondierenden Zahl von `0` bis `7`.

In meiner `Cube`-Klasse erstellte ich nun eine Definition, welche die einzelnen Ecken darstellen kann. Die Definition sieht wie folgt aus:

```
def display(self):
        pygame.draw.rect(screen, colors[self.colors[0]], [self.pos[0][0],
self.pos[0][1], 50, 50])
        pygame.draw.rect(screen, colors[self.colors[1]], [self.pos[1][0],
self.pos[1][1], 50, 50])
        pygame.draw.rect(screen, colors[self.colors[2]], [self.pos[2][0],
self.pos[2][1], 50, 50])
```

In dieser Definition zeichne ich dreimal ein Rechteck auf der im Voraus erstellten Oberfläche `screen` mit den Farben, die ich dem `cubicle` gebe. Das Ganze geschieht an den Koordinaten x und y. Die Grösse des Rechtecks ist 50 mal 50 Pixel.

Die zweite und letzte Definition in der `Cube` Klasse nannte ich `move(self)`, denn die Definition ist für die Bewegung der Kacheln verantwortlich. Als erstes wird abgefragt, welche Seite gedreht wird. Dies wird im Game-Loop weiter unten im Code durch Tastenabfrage bestimmt. Wenn zum Beispiel die obere Seite `'u'` gedreht wird, überprüfe ich nun für alle möglichen Koordinaten auf der oberen Seite, ob sie sich an einer bestimmten Position befinden. Wenn dies der Fall ist, werden die x- und y-Koordinaten so abgeändert, dass sie sich an der neuen Position befinden.

Nachdem ich dies auch für die Rotation `'f'` und `'d'` gemacht hatte, stiess ich auf ein grösseres Problem. Mit der aktuellen Methode müsste ich für jede Ecke an jeder Position, in

jeder Orientierung eine «Wenn, dann, sonst…»-Abfrage machen. Da ich 8 `cubicles` mit jeweils 8 möglichen Positionen und 3 möglichen Orientierungen habe, wobei jede Überprüfung 4 Zeilen Code benötigt, ergäbe dies insgesamt einen 768-zeiligen Code nur für alle Drehmöglichkeiten. Da mir dies zu aufwändig war, beendete ich an der Stelle Version 1 und startete ein neues Dokument.

Beim Endpunkt der Version 1 konnte diese bereits eine komplette Drehung der oberen (weissen) Seite und der unteren (gelben) Seite vornehmen. Zusätzlich konnte sie eine 90°-Drehung der vorderen (grünen) Seite vornehmen. In Kombination mit einer Drehung oben oder unten konnten die 2 Teile, die von der vorderen Seite auf die obere bzw. untere Seite gedreht wurden, nicht weitergedreht werden, da ich diese Koordinatenkombination nicht manuell hinzufügte.

Ansonsten initialisierte ich im Code nur noch alle weiteren `cubicles`, zeichnete Linien zwischen den einzelnen Kacheln und führte Tastenabfragen durch. Der Cube im Ursprungszustand sieht wie folgt aus:

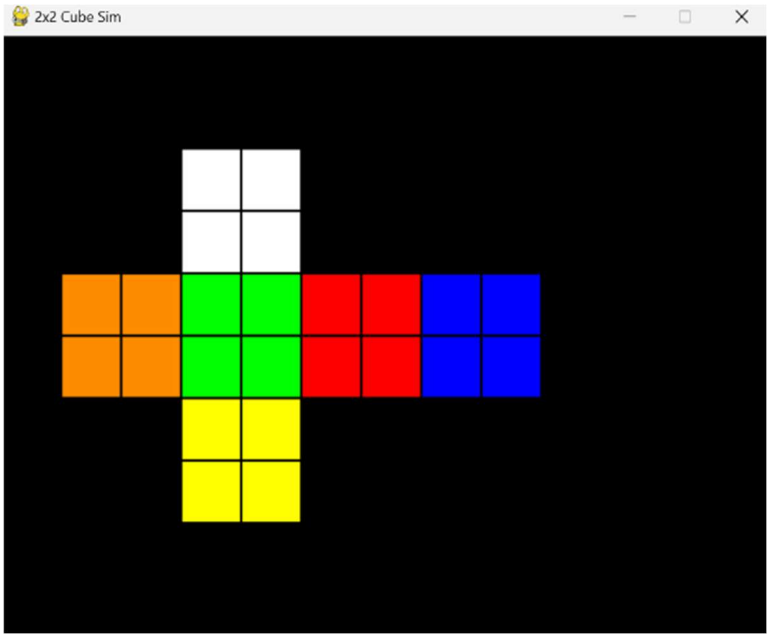


Abb. 2: 2D-Cube Version 1

4.3. Version 2

In meiner zweiten Version versuchte ich einen 2D-Cube zu erstellen, der jedoch 3D aussieht, nicht beweglich ist, aber trotzdem alle Farben sichtbar darstellt. Ein Beispiel nahm ich mir von der Website *alg.cubing.net*[7], die es so darstellt, wie ich es mir vorstellte.

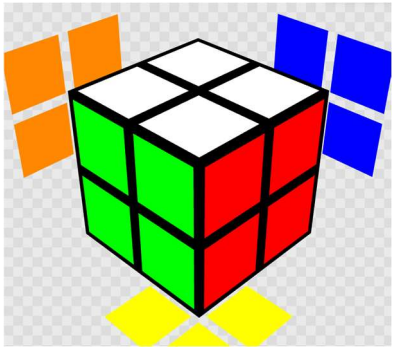


Abb. 3: Inspiration von alg.cubing.net

Ich definierte wieder alle Positionen der einzelnen Kacheln und trug alles manuell ein. Der einzige grössere Unterschied zur ersten Version war, dass ich nicht mehr Rechtecke, sondern Polygone zeichnete. Um in Pygame ein Polygon zu zeichnen, muss man für alle Ecken des Polygons die x- und y-Koordinate angeben. Diese Punkte werden dann verbunden und mit einer ausgesuchten Farbe gefüllt. Ich vereinfachte das Erstellen eines Eckteils, indem ich die ausgeschriebenen Koordinaten in der Klasse berechnete, anstatt sie direkt mitzugeben.

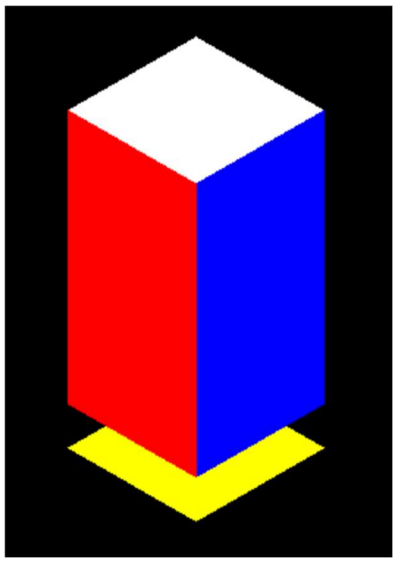


Abb. 4: Beginn eigener Cube, Version 2

[7] https://alg.cubing.net/?puzzle=2x2x2

Als ich realisierte, dass ich mit meiner aktuellen Darstellungsstruktur wieder genau das gleiche Problem haben werde, wie in Version 1, brach ich auch diese Version 2 ab. Da ich auch hier alle Koordinaten manuell eintippte, müsste sie dementsprechend auch wieder manuell neu berechnet werden.

4.4. Version 3

Für meine dritte Version wagte ich mich an eine korrekte 3D-Repräsentation eines Würfels. Da ich selbst noch nie eine 3D-Grafik erstellt hatte, suchte ich mir dafür ein Tutorial (A., 2021). Ich fand ein 24-minütiges YouTube-Video[8] des Kanals *Yuta A*[9]. Das Video erklärt Rotationsmatrizen, Projektion und wie man diese Berechnungen im Pygame-Fenster anzeigen und rotieren kann. Am Ende des Tutorials hatte ich nun einen rotierbaren Quader in Pygame.

Das Programm berechnet die 8 Eckpunkte und verbindet diese danach mit Linien. Gesteuert wird der Cube mit den Tasten W, A, S, D, Q und E. Um die Eckkoordinaten zu berechnen, verwendet das Programm Matrizen. Diese Matrizen werden in einer Funktion miteinander multipliziert. Nach einer Reihe von Mulitplikationen entsteht eine Matrix, deren x- und y-Koordinaten als Koordinaten eines Punktes verwendet werden. Danach werden die Punkte in einer weiteren Funktion miteinander verbunden.

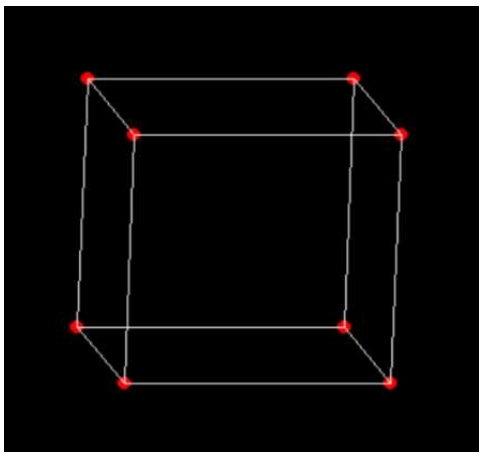


Abb. 5: Cube aus dem Tutorial

Ich war zufrieden mit dem Resultat. Deshalb entschied ich mich für die nächste Version, den Cube in eine Klassenstruktur zu packen und ihn so anzupassen, damit ich die Klasse erweitern kann und so eine bessere Übersicht habe.

[8] https://www.youtube.com/watch?v=sQDFydEtBLE
[9] https://www.youtube.com/@yutaa.1904

4.5. Version 4

Mein Ziel für die vierte Version war, das Tutorial selbst so umzuschreiben, dass das Programm zwar noch dasselbe kann, jedoch mit einer Klassenstruktur. Dies erreichte ich, indem ich die Rotationskalkulation in eine Definition in der Klasse verschob. Ab Version 4 verwendete ich zudem erstmals Numpy. Der Hauptgrund dafür war, dass ich mit Numpy Koordinaten und Vektoren nicht mehr als Liste, sondern als Numpy-Array speichern konnte. Das Problem mit Listen ist, dass schnell ein Referenzfehler auftreten kann. Solche Fehler sind schwer zu finden. Ebenfalls kann man mit Numpy-Arrays viele Berechnungen mit Numpy-Funktionen durchführen, statt die Formeln selbst eintippen zu müssen.

Das Programm stellte wie das Tutorial nur 8 Eckpunkte und die verbindenden Linien dar. Die Koordinaten eines Eckpunktes wurden als 3D-Vektoren gespeichert, die auf allen Achsen den Wert 1 haben, jedoch mit variierenden Vorzeichen. Wenn man alle Kombinationen davon in ein Koordinatensystem einzeichnet, bekommt man die 8 Eckpunkte eines Würfels.

Das Programm funktionierte genauso, wie ich es wollte, d. h., der Cube konnte im Fenster rotiert werden, damit man einen neuen Ansichtswinkel bekommt. Dann wechselte ich den zuvor genannten Vektor von einer Liste zu einem `np.array()`. Nachdem ich die Speicherung der Koordinaten auf Arrays änderte, verschwand der Cube plötzlich nach Betätigen von 4 Pfeiltasten.

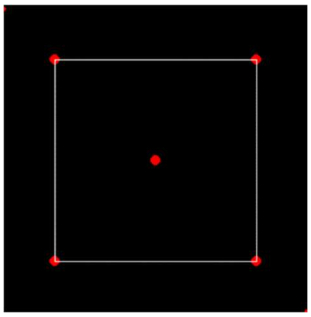


Abb. 6: Vor dem Tastendruck

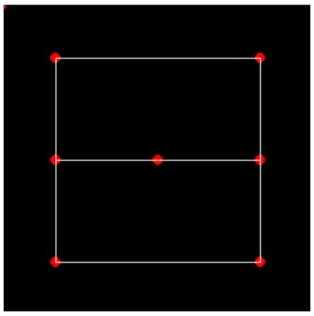


Abb. 7: Tastendruck 1

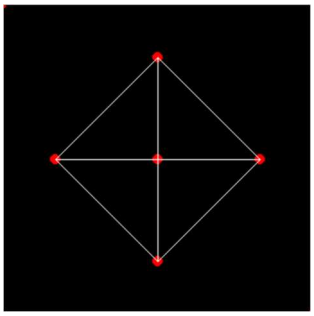


Abb. 8: Tastendruck 2

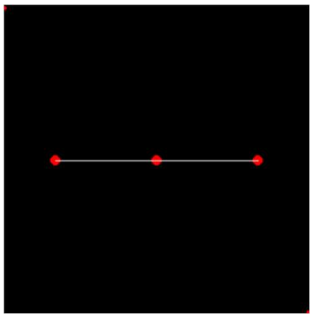


Abb. 9: Tastendruck 3

Abb. 10: Tastendruck 4

Auf den Abbildungen 6 bis 10 sieht man, wie der Würfel immer kleiner wird und schliesslich nur noch ein einzelner Punkt dargestellt ist. Ich begab mich auf Fehlersuche.

Ich druckte zuerst an verschiedenen Stellen im Code den Vektor aus. So fand ich heraus, dass er bei der Rotationsberechnung Stück für Stück auf `(0,0,0)` gesetzt wird. Mithilfe des Debug-Tools entdeckte ich, dass die Neuberechnung des Vektors ihn auf 0 setzte. Ich führte dieselben Berechnungen in einem separaten Programm nochmals durch, bekam jedoch Vektoren wie z. B. `[-0.85765159 -1.14412281  0.97745423]`. Doch in diesem separaten Programm verwendete ich eine Variable anstelle eines Numpy-Arrays. Als ich die Variable auf `np.array()` änderte, ergab es den Vektor `[0,0,0]`! Nun wusste ich also definitiv, dass der Fehler beim Array lag.

Da fiel es mir wie Schuppen von den Augen. Beim Initialisieren des Arrays verwendete ich `[1,1,1]`. Dies sind drei Int's, das heisst, sie speichern ganze Zahlen. Jedoch berechne ich einen Float, das heisst eine Kommazahl! Die Numpy-Arrays behalten aber den Datentyp, den man ihnen am Anfang gibt. In meinem Fall einen Int. Dies bedeutet, dass die Koordinaten langsam, aber sicher zu `0` abgerundet werden, weshalb mein Cube zum Nullpunkt hin verschwindet. Um diesem Problem entgegenzuwirken, änderte ich die Initialisierung zu `[1.0,1.0,-1.0]`. Das Problem war sofort gelöst und der Cube wieder bewegbar.

Ich speicherte die fehlerhafte Version so wie sie vorher war ab, damit ich dieses spezielle Problem dokumentiert habe.

4.6. Version 5

Ich fuhr dort weiter, wo ich bei Version 4 aufgehört hatte, jedoch in einem neuen Dokument. Mein Ziel war es nun, innerhalb der Klasse eine Trennung der 8 Eckstücke einzufügen. Einmal mehr verwendete ich weitere Numpy-Arrays, die vom Eckpunkt aus in 3 Richtungen zeigen.

Den Startvektor, von dem alle weiteren Vektoren berechnet werden, nannte ich `vec`, den Problemvektor aus Version 4. Vom Endpunkt des Vektors `vec` aus starten nun 3 weitere Vektoren, einer auf jeder Achse. Den Vektor auf der x-Achse nannte ich `vecx`, auf der y-Achse `vecy` und auf der z-Achse `vecz`. Um den vierten Punkt der Polygone zu berechnen, erstellte ich einen Vektor, der vom Nullpunkt aus gerade auf der korrespondierenden Achse verläuft.

Den Vektor auf der x-Achse nannte ich `p4`, den auf der y-Achse `p5` und den auf der z-Achse `p6`.

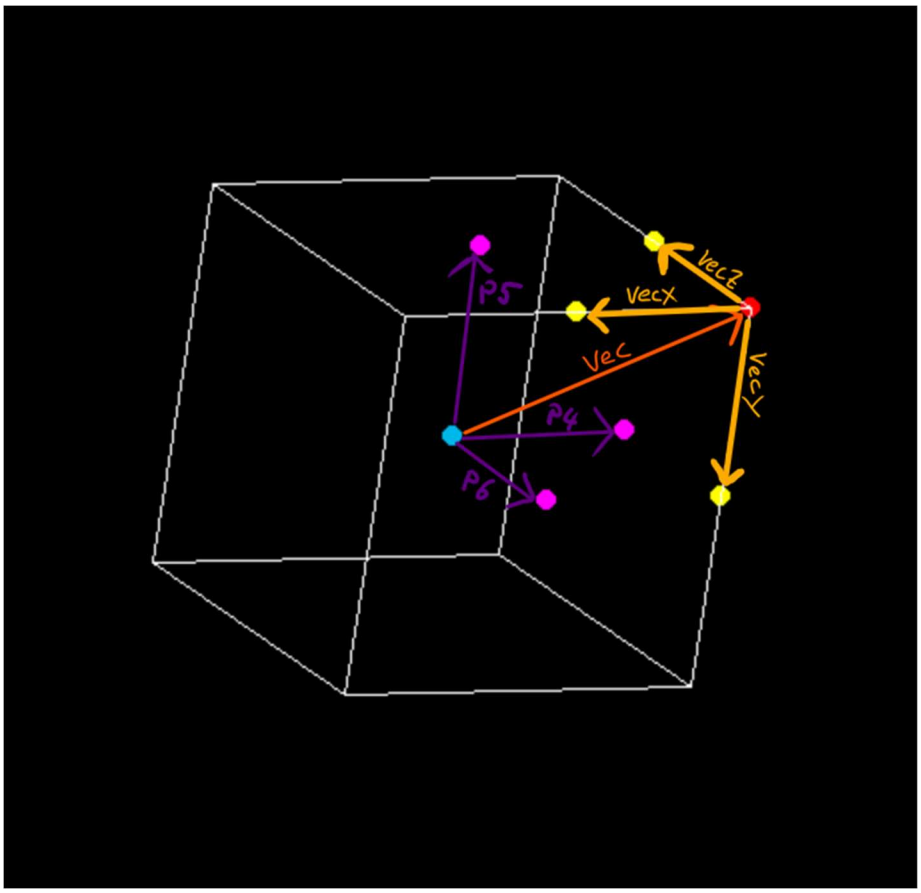


Abb. 11: Benennung der Vektoren aus Version 5

Auf Abbildung 11 sind alle Vektoren des Eckteil Nummer 6 eingezeichnet und beschriftet. Die Länge aller Vektoren ist 1, und zwischen den Endpunkten der Vektoren konnte ich nun die gewünschte Farbe einfüllen und bekam so ein Eckteil.

Da die abgespeicherten Vektoren eine Grösse von 1 hatten, multiplizierte ich die Endkoordinaten immer mit 100, um eine ansprechendere Grösse zu erhalten. Weiter addierte ich zu den Koordinaten noch 300, um die Koordinaten in die Mitte des Displays zu verschieben.

Ich schrieb eine einfache Definition in der Cube-Klasse, die genau wie in den vorherigen Versionen, Polygone berechnete und danach mit der ausgesuchten Farbe füllte. Das Problem dabei war, dass ich kein sogenanntes Z-Buffering codierte. Dies heisst, es gab keine Tiefe im Cube. Man muss sich das so vorstellen: Wenn ein Polygon mit einer Farbe gefüllt wird, klebe

ich ein Post-it auf ein Blatt Papier. Jedes weitere Post-it wird darüber geklebt. Das Ganze sah ziemlich chaotisch aus…

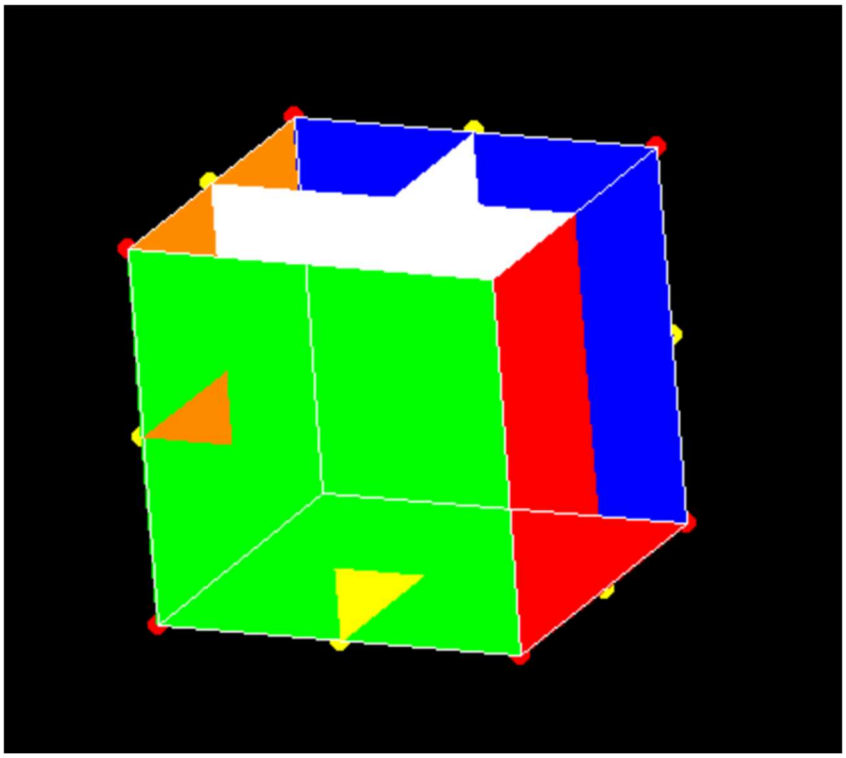


Abb. 12: Cube Version 5, ohne Z-Buffering

Hier sollten z. B. nur die Farben Weiss, Grün und Rot sichtbar sein. Da aber gewisse blaue, orange und gelbe Felder erst nachher gezeichnet werden, werden sie darüber gezeichnet. Ich wusste noch nicht, wie ich das Problem lösen sollte. So versuchte ich also zuerst den Drehalgorithmus zu erstellen, damit ich im Hinterkopf noch weiter am Z-Buffer überlegen konnte.

Um eine Drehung einer beliebigen Seite vorzunehmen, muss ich alle vorher erstellten Numpy-Arrays, die sich auf der zu rotierenden Seite befinden, um die gewünschte Achse rotieren. Dies bedeutet, dass ich wissen muss, welche Vektoren zu rotieren sind und um welche Achse dies geschehen soll. Also erstellte ich eine weitere Definition in der Cube-Klasse. Die Definition nannte ich `turn()`, und ich gab den Parameter `turn` mit. Die Variable `turn` beinhaltet einen Buchstaben, der in der offiziellen WCA-Notation des 2x2-Zauberwürfels formatiert ist, und aussagt, welche Seite gedreht wird. Als erstes implementierte ich die Seite `r`, also rechts.

In `turn()` überprüfte ich nun, ob `vec` des Teiles, welches ich drehen will, rechts von der x-Koordinate 0 liegt. Ich kann dies tun, indem ich überprüfe, ob die x-Koordinate grösser als 0 ist. Wenn dies nicht der Fall ist, passiert nichts, andernfalls aber rotiert das Teil. Ich suchte nun

im Internet nach einem Algorithmus, der mir ermöglicht, einen Vektor rund um die x-Achse zu drehen. Ich fand keinen. Die Formeln, die ich fand, waren nicht in Python geschrieben, verwendeten komplexe Bibliotheken, oder ich wusste nicht, wie ich sie in Python verwenden konnte. Also fragte ich kurzerhand ChatGPT, ob es mir eine Python-Definition schreiben könnte, die einen 3D-Numpy-Array um die x-Achse drehen kann. ChatGPT brauchte ein paar Versuche, bis die Funktion funktionierte, doch im Endeffekt hatte ich einen Algorithmus, der das tat, was ich wollte. Als ich ihn dann jedoch ausprobierte, drehte die Seite nicht um 90°, und die gedrehte Seite war schräg.

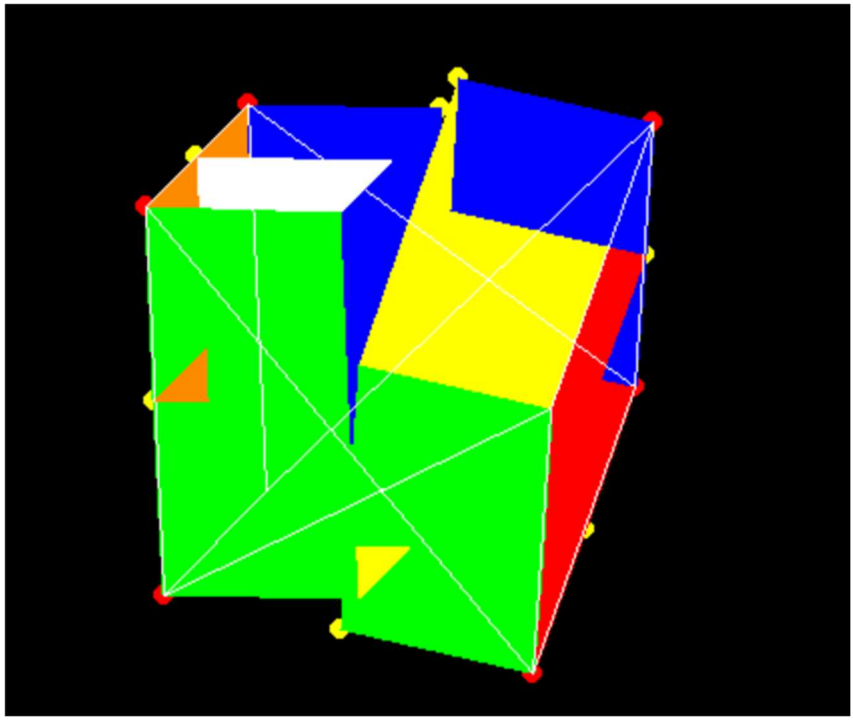


Abb. 13: Inkorrekt gedrehte Seite rechts, Version 5

Nun hatte ich zwei Probleme: das nicht vorhandene Z-Buffering und die schräg gedrehte Seite. Zuerst fand ich heraus, dass der Winkel nicht in Grad angegeben werden sollte, sondern in Bogensekunden. Dies fand ich heraus, da bei meiner Recherche nach Rotationsalgorithmen mehrere Quellen Bogensekunden verwendeten. Um dies zu konvertieren, verwendete ich die Numpy-Funktion `np.radians(90)`. Nun drehte sich die Seite um 90°, war jedoch immer noch schräg. Nach Ausprobieren im Programm fand ich heraus, dass das Programm genau das machte, was ich ChatGPT als Auftrag gab. Es drehte nämlich alle Vektoren um die x-Achse. Wenn ich aber nun den Cube im Raum bewegte, sollte er nicht mehr um die x-Achse gedreht werden, sondern um eine gleich wie der Cube, veränderte Achse. Da ich auch diesen Fehler dokumentieren und lösen wollte, erstellte ich eine neue Python-Datei.

Im Verlauf dieser Version war es mir zu umständlich geworden, für jede Rotation im Raum und jede Drehung die 8 einzelnen Cubes aufzurufen. Deshalb programmierte ich dafür einen kleinen Algorithmus, der alle 8 Cubes aufruft und z. B. eine Drehung an der rechten Seite vornimmt.

4.7. Version 6

Ich duplizierte Version 5, benannte sie neu und löste als Erstes mein Problem mit der Rotation. Ich fragte ChatGPT, wie ich einen 3-dimensionalen Vektor um einen zweiten 3-dimensionalen Vektor rotieren kann. Als Antwort erhielt ich eine ausführliche Erklärung der Rodrigues Formel, die verwendet werden kann, um Vektoren um Vektoren zu Rotieren. Danach listete es auf, welche Werte man haben muss, um die nachher aufgeführte Formel nutzen zu können. Die Formel[10] war bereits in x-, y- und z-Achse aufgeteilt. Die 3 Formeln konnte ich so übernehmen und musste nur noch die richtigen Variablen dazu erstellen. Dann funktionierte es.

Damit ich diese Funktion nun nicht für jede der 6 Seiten kopieren und wieder einfügen musste, erstellte ich eine Definition, genannt `rot()`, ausserhalb der Cube-Klasse, die diese Rotation berechnet. Der Funktion muss der zu rotierende Vektor die Achse, um die rotiert wird und der Rotationswinkel mitgegeben werden. In der Funktion wird die Achse zu einem Normalvektor gemacht, der Winkel in Bogensekunden transformiert und danach werden die neuen x-, y- und z-Koordinaten berechnet und zurückgegeben.

Die `turn()` Funktion in der Klasse behielt ich bei. Diese änderte ich aber so ab, dass sie für jeden Vektor, der rotiert werden sollte, die Funktion `rot()` aufruft. In `turn()` berechnete ich ebenfalls die Achse, um die rotiert wird. Diese ist entweder `p4`, `p5` oder `p6`. Sie verlaufen am Anfang auf der x-, y- oder z-Achse, nur dass sie mit dem Cube mitrotiert werden. Bleiben wir beim Beispiel der Drehung `'r'`. Um zu bestimmen welchen der 3 Vektoren ich benötige, erstellte ich eine leere Liste und fügte dort die x-Koordinate der 3 Vektoren an. Danach sortierte ich die Liste der Grösse nach. Die Achse, die den grössten x-Wert hat, verwendete ich dann um alle Vektoren darum rotieren zu lassen.

Nun implementierte ich dieses System auch für die anderen 5 Drehungen und führte auch die dazugehörigen Tastenabfragen durch.

[10] Formel von ChatGPT im Code auf Zeile 323-325

4.7.1. Z-Buffer

Inzwischen hatte ich eine Idee, wie ich das Z-Buffering einbauen könnte. Ich erstellte eine neue Definition genannt `buffer()`. In dieser Funktion erstellte ich eine neue Liste und fügte von allen 8 Cubes den z-Wert von `vec` in eine untergeordnete Liste ein. In diese zweite Liste kam ebenfalls die Nummer des Cube. Ich sortierte die Listen in der Liste nach dem z-Wert. Nun rief ich zuerst den hintersten Cube auf, damit er mit Farbe gefüllt wird. Danach folgte die Funktion der Reihenfolge der sortierten Liste, bis alle 8 Cubes gefüllt waren. Der Vorderste wurde zuletzt gefüllt, das heisst, er wurde auch zuvorderst angezeigt.

Nun war die Reihenfolge, in welcher die 8 Cubes angezeigt wurden, zwar korrekt, aber innerhalb eines Teiles spielte es auch noch eine Rolle, welches Polygon zuerst angezeigt wird. Ein Polygon kann sich nämlich hinter den beiden anderen verstecken. Also bearbeitete ich die `fill()` Funktion in der Cube-Klasse sodass innerhalb des Teiles nochmals sortiert wird, welches der 3 Polygone das Hinterste ist. Dieses muss folglich zuerst gezeichnet werden. Dafür verwendete ich den genau gleichen Code wie bei `buffer()`, aber anstelle der acht z-Koordinaten verwendete ich die 3 z-Koordinaten der Vektoren `p4`, `p5`, und `p6`.

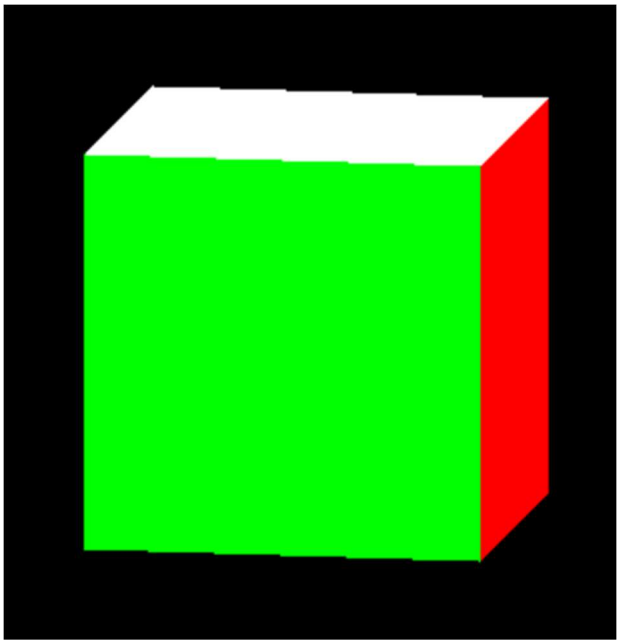


Abb. 14: Cube aus Version 6, funktionierendes Z-Buffering

Mit dem Programm war es mir an dieser Stelle möglich den Cube zu verdrehen und wieder zu lösen. Doch im Programm selbst fand noch keine Überprüfung statt, ob der Cube gelöst ist oder nicht. Dies war mein nächstes Ziel.

Ich erstellte wieder eine neue Definition, genannt `checker()`. Sie kontrolliert, ob der Cube gelöst ist. Zuerst hatte ich die Idee zu prüfen, ob die Eckpunkte alle an der richtigen Position sind. Ich kann jedoch den Würfel im Raum drehen und so die Position ändern. Weiter kann ich ihn auch in andere gelöste Zustände bringen, wo nicht Grün vorne und Weiss oben ist. Als Lösung fand ich die Methode zu kontrollieren, ob die richtigen Teile miteinander benachbart sind. So überprüfte ich mit `np.allclose()`, ob sich z. B. zwei `vecx`, die aufeinander zeigen, auf `0.00001` Pixel genau die gleichen Koordinaten haben. Wenn dies der Fall war, tat ich dies für die nächsten zwei Vektoren. Mein System, um auszusuchen, welche zwei Vektoren ich überprüfen sollte, verbindet alle Teile miteinander. So kann ich garantieren, dass alle Teile an der richtigen Position sein müssen und die richtige Orientierung haben müssen. War dies bei allen sieben Überprüfungen der Fall, setzte ich die Variable `state` auf `True`. War jedoch schon ein einziges Vektorpaar nicht benachbart, setzte ich `state` auf `False`.

Ich wollte diese Abfrage noch visualisieren, deshalb zeichnete ich in der oberen linken Ecke entweder einen grünen Haken oder ein rotes X. Ich verschob das Ändern der Variable `solved` ebenfalls in die Zeichnungsfunktion. Dadurch vermeide ich, dass der Status des Cube als gelöst gilt, jedoch grafisch nicht angezeigt wird.

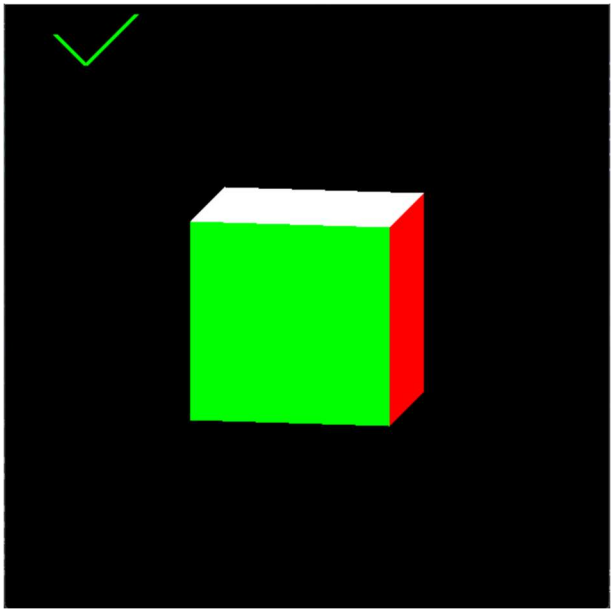


Abb. 15: Version 6, Cube gelöst

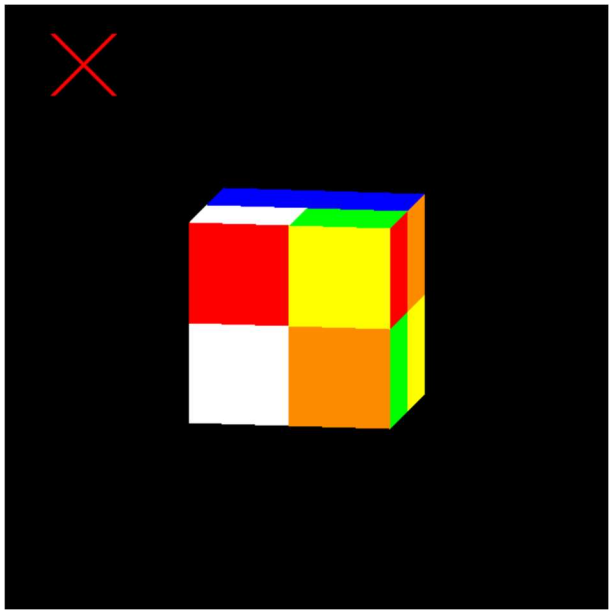


Abb. 16: Version 6, Cube nicht gelöst

4.7.2. Zufallsalgorithmus

Endlich konnte ich beginnen, einen Lösungsalgorithmus zu programmieren. Mein erster Lösungsalgorithmus sollte ein Zufallsgenerator sein, welcher so lange eine zufällige Drehung macht, bis der Cube gelöst ist. Dafür erstellte ich eine `while` Schleife, die so lange läuft, bis `state == True` erfüllt ist. In dieser Schleife erstellte ich zwei Zufallsvariablen. Die erste bestimmt eine der 6 Seiten, die gedreht wird, die zweite bestimmt, in welche Richtung die vorher bestimmte Seite gedreht wird. Nun wird diese Drehung ausgeführt und geprüft, ob der Cube gelöst ist. Wenn dies nicht der Fall ist, wählt das Programm wieder eine zufällige Seite und Richtung zum Drehen. Falls der Würfel nun gelöst ist, gibt das Programm die benötigte Zeit und die benötigte Anzahl Drehungen in der Konsole aus.

Während der Cube gelöst wurde, konnte ich nicht mit dem Fenster interagieren. Das hatte zur Folge, dass Windows nach kurzer Zeit anzeigte, dass das Programm nicht reagiert. Deshalb baute ich noch Tastenabfragen in diese Schleife ein, damit ich das Programm wahlweise anhalten oder beenden kann. Anschliessend implementierte ich eine Möglichkeit, dass nach dem Lösen ein nächster Cube verdreht und wieder gelöst wird. Dieser Loop wird durch die Taste `W` (Wiederholung) aktiviert, und gleichzeitig durch ein `L` (Loop) oder `N` (Not Loop) in der linken oberen Ecke visualisiert. Weiter ermöglichte mir dieser Loop, das Programm über Nacht laufen zu lassen, um Daten zu sammeln. Wenn der Cube gelöst ist, wird das Endresultat nicht nur in die Konsole geschrieben, sondern auch in eine `.txt` Datei, die sowohl alle Zeiten als auch Anzahl Drehungen sammelt. Ich fügte auch noch ein Display-Update ein, damit die Bewegung ebenfalls sichtbar ist.

4.7.3. Py222

Als zweiten Lösungsalgorithmus suchte ich online nach einer Bibliothek, die ich importieren kann, die meinen verdrehten Würfel effizient löst. Ich fand mehrere Bibliotheken und Programme zum Importieren; Alle hatten jedoch eine Farbcodierung als Input benötigt, die ich aber nicht generiere. Also suchte ich weiter. Ich fand eine GitHub-Seite[11] (Asis, 2017), genannt Py222, die ein Programm enthielt, in welchem man nicht den Farbcode als Input gibt, sondern den Verdreh-Algorithmus in WCA-Notation. Py222 verwendet ebenfalls die Numpy-Bibliothek.

`Py222` ist ein Python-Programm, welches einen 2D-Cube in der Konsole abbilden kann. Mit dem Befehl `py222.doAlgStr(cube, 'U2 F2 R2 U2')` kann man den Cube verändern. Der Beispielalgorithmus macht ein einfaches Muster. Auf GitHub ist eine zweite Python-Datei, genannt `solver`, die ich ebenfalls benutzte. Diese Datei ermöglicht es, einen verdrehten Cube zu lösen. Ich verstehe nicht komplett, wie das Programm funktioniert. Es gibt viele Variablen, Abkürzungen und Funktionen, deren Namen sehr kurz und mir unverständlich sind. Ebenfalls sind gewisse Funktionen programmübergreifend, was zusätzlich Verwirrung schafft. Es gibt auch keine Dokumentation, die den Code erklärt und die Kommentare im Code sind sehr kurz. Die Dokumentation[12] zeigt, welche Schritte man benötigt, damit das Programm funktioniert. Die grauen Texte beschreiben, was unter ihnen passiert. Es sind die folgenden Schritte:

```
# Die beiden Programme importieren
import py222
import solver

# Einen Cube genannt myCube erstellen, und in Grundposition definieren
Cube = py222.initState()

# Der Cube führt nun den folgenden Bewegungsablauf durch
Cube = py222.doAlgStr(Cube, "R U2 R2 F2 R' F2 R F R")

# Der Cube wird vom solver-Programm gelöst
solver.solveCube(Cube)
```

[11] https://github.com/MeepMoop/py222
[12] https://github.com/MeepMoop/py222#solver-usage

Dieser Code gibt Folgendes in die Konsole aus:

```
      ┌──┬──┐
      │ 2│ 3│
      ├──┼──┤
      │ 1│ 0│
      └──┴──┘
┌──┬──┬──┬──┬──┬──┬──┬──┐
│ 1│ 3│ 5│ 4│ 2│ 2│ 4│ 3│
├──┼──┼──┼──┼──┼──┼──┼──┤
│ 4│ 2│ 0│ 0│ 4│ 1│ 0│ 5│
└──┴──┴──┴──┴──┴──┴──┴──┘
      ┌──┬──┐
      │ 1│ 5│
      ├──┼──┤
      │ 3│ 5│
      └──┴──┘
normalizing stickers...
generating pruning tables...
searching...
depth 1
depth 2
depth 3
depth 4
depth 5
depth 6
depth 7
depth 8
F R2 F' R U2 R2 F' R
F R2 F' R' F R2 U2 R'
```

Das obere 2D-Layout des Cube stellt den Cube im verdrehten Zustand dar. Die Zahlen stehen für die verschiedenen Farben. 0 ist weiss, 1 ist rot, 2 ist grün, 3 ist gelb, 4 ist orange und 5 ist blau. Das Programm sucht nun eine Lösung, die nur eine Drehung benötigt. Wenn das Programm keine findet, sucht es nach einer Lösung, die 2 Drehungen benötigt. Solange es keine findet, sucht es weiter mit einer zusätzlichen Drehung. Dies wird mit `depth 1` bzw. `depth 2` … angegeben. Bei einer Tiefe von 8 Drehungen hat das Programm nun 2 Lösungen gefunden. Diese unterscheiden sich nur wenig voneinander, kommen aber zum selben Resultat. Beide benötigen eine Drehung weniger, um den Cube zu lösen, als benötigt wurde, um den Cube zu verdrehen.

Damit ich diese Programme jedoch verwenden konnte, musste ich ein paar Dinge anpassen. Das erste offensichtliche Problem war, dass die direkt heruntergeladene Version nicht

funktionierte. Die Ursache dafür war schnell gefunden, denn in der Konsole wurde der unbekannte Ausdruck und die Zeile genannt. Im Code wurde die Funktion `np.int` verwendet, um eine Zahl als `int` zu definieren. Jedoch gibt es diesen Befehl nicht mehr. Ich löschte also kurzerhand den `np.`-Teil weg, denn `int` ist eine in Python definierte Funktion. Es funktionierte. Aber an anderen Stellen kam nochmals dasselbe Problem vor, weshalb ich alle Instanzen von `np.int` in beiden Programmen zu `int` änderte. Als nächstes musste ich eine Schnittstelle zwischen meinem Programm und `Py222` erstellen. `Py222` verwendet die WCA-Notation in Grossbuchstaben, ich verwendete jedoch Kleinbuchstaben. Ich änderte die Funktion `scramble` so ab, dass sie die ausgesuchte Drehung und die Drehrichtung in `Py222`-lesbarer Formatierung in eine Liste speichert.

Ich erstellte nun in meinem Programm eine neue Definition, genannt `solve222()`, die den Cube mithilfe von `Py222` löst. Aufgerufen wird `solve222()` mit der Taste 3. Der Cube wird ganz am Anfang initialisiert. Innerhalb von `solve222()` wird nun der Scramble angewandt und die `solve`-Funktion aufgerufen. Ich passte die `solve`-Funktion so an, dass alle möglichen Lösungen in eine Liste gespeichert werden. Zwischen den einzelnen Lösungsalgorithmen wird mein Schlüsselwort `'gap'` eingefügt. In `solve222()` wird nun alles nach dem Schlüsselwort `'gap'` aus der Liste entfernt und alle Buchstaben werden zu Kleinbuchstaben. Danach wird dieser Algorithmus ausgeführt. Am Schluss werden auch hier die benötigte Zeit und die Anzahl Drehungen in eine `.txt`-Datei gespeichert. Ich konnte diese Funktion leider nicht mehrmals wiederholen. `Py222` mit meinen Ergänzungen ist dafür ausgelegt, nur einen Cube zu lösen. Für einen weiteren Versuch muss ein Neustart erfolgen. Ich verstehe das Programm zu wenig, um dies selbst einzubauen. Damit bei mir die Konsole nicht zu voll wird, setzte ich vor jede `print`-Funktion ein `#`, damit nichts ausgedruckt wird.

4.7.4. Optimierter Zufallsalgorithmus

Nachdem ich nun einen effizienten Weg, den Cube zu lösen im Internet gefunden und kopiert hatte, wollte ich selbst auch noch einen effizienteren Algorithmus programmieren. Dafür erstellte ich eine Funktion, genannt `solveR2()`, welche den Cube zufällig, aber effizienter löst. Ich kopierte meine Zufalls-Lös-Funktion und passte einige Codeelemente an.

Als Erstes änderte ich die Auswahl der nächsten Drehung. Ich speicherte, was die letzte Drehung war. Die nächste Drehung kann nicht dieselbe Seite drehen und auch nicht die gegenüberliegende Seite. Dies tat ich mit mehreren einfachen `if lastTurn == 'turn'` Abfragen.

Nun wollte ich noch ein System programmieren, welches detektiert, wenn der Cube eine Drehung vom gelösten Zustand entfernt ist und die korrekte Drehung findet. Damit ich dies detektieren kann, verwendete ich einen ähnlichen Algorithmus wie für die Funktion `solved()`. Ich überprüfte für zwei gegenüberliegende Seiten, ob sie beide gelöst sind. Dies geschieht für alle 3 möglichen Seitenpaare. Wenn zwei Seiten komplett gelöst sind, setzte ich die Variable `oma`, welche 'one move away' bedeutet, auf `True`. Nun wird nach jeder Drehung dieser `oma`-Check durchgeführt und falls `oma == True` ist, wird die richtige Drehung gesucht. Damit die richtige Drehung gefunden wird, wird zuerst die obere Seite `'u'` um 90° gedreht und überprüft, ob der Cube nun gelöst ist. Falls nicht, wird um weitere 90° gedreht und wieder überprüft, ob der Cube gelöst ist. Dies geschieht insgesamt 4x. Danach wird dasselbe für die rechte Seite `'r'` und die vordere Seite `'f'` repetiert. Bei der Drehung, welche den Cube löst, hört dieser Drehalgorithmus auf; es wird keine weitere Drehung mehr ausgeführt, denn der Cube ist ja gelöst. Ich versuchte diesen Algorithmus noch zu optimieren, fand jedoch keinen Weg dafür. Meine Idee war noch, dass dadurch, dass ich weiss, welche zwei Seiten gelöst sind, ich nur auf einer Seite 4 Drehungen ausprobieren muss. Mir fiel aber auf, dass dies nicht funktionieren würde, denn die weisse Seite ist z. B. nicht immer oben, sondern kann auch mal vorne sein. Somit würde 4x die obere Seite drehen, den Cube doch nicht lösen. Man könnte zwar berechnen, welches die korrekte Seite wäre, jedoch habe ich bereits an einer anderen Stelle im Code eine solche Berechnung, die leider ein grösseres Problem verursacht.

4.7.5. Fehler

Dieses Problem entsteht beim Drehen einer Seite in einer speziell rotierten Position. Die Problematik lautet wie folgt: Wird der Cube mithilfe der Pfeiltasten in eine Schräglage bewegt, kann es sein, dass die falschen Teile um die falsche Achse rotiert werden. Nehmen wir das Beispiel, dass ich die rechte Seite `'r'` drehen will. Der Cube sieht aktuell wie folgt aus:

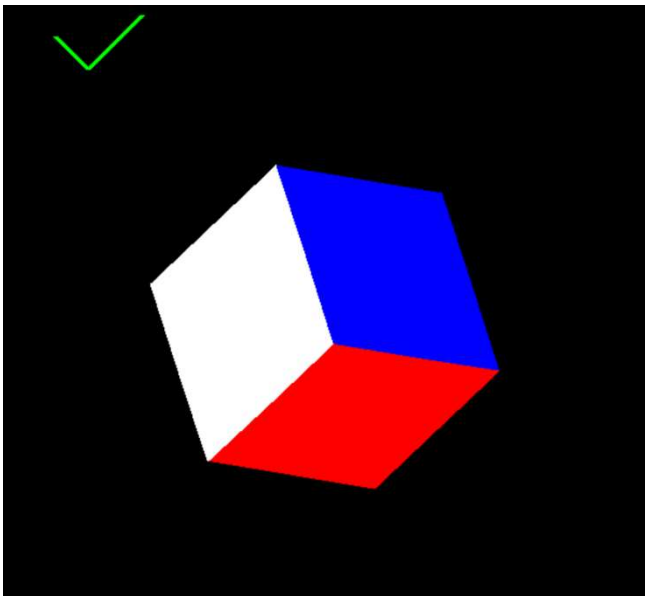


Abb. 17: Cube normal, ungünstige Rotation, Version 6

Man kann verstehen, dass es schwierig zu definieren ist, welche der 8 Teile bei der Drehung ‘r’ rotiert werden sollten. In meinem Code wird dies bestimmt, indem überprüft wird, ob sich der Eckpunkt `vec` rechts der Mittellinie befindet. Wenn dies der Fall ist, wird dieser Punkt um die Achse gedreht, die am nächsten an der x-Achse ist. Dies alles führt zu diesem Cube:

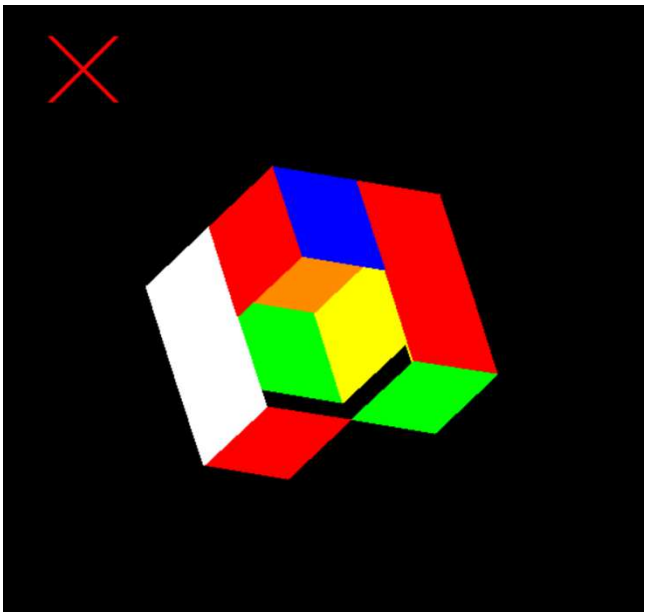


Abb. 18: Cube nach Rotation 'r', Fehler, Version 6

Die Ursache dafür ist, dass das einzige Kriterium, um ein Teil zu rotieren, ist, dass es sich links bzw. rechts der Mittellinie befindet. Ich habe noch keine Lösung gefunden, da ich selbst nicht weiss, welche Drehung ich auf Abb. 17 als ‘rechts’ betiteln würde. Wenn ich mir als Nutzer der Problematik bewusst bin, ist sie vermeidbar; im Zweifelsfall dreht man den Cube noch ein bisschen weiter auf die *sichere* Seite.

Ich fügte noch kleinere Details in diese Version ein. Dies beinhaltet unter anderem eine Anleitung beim Starten des Programmes in der Konsole und ein wenig Rotation des Cube, damit man direkt einen 3D-Würfel sieht und nicht nur die grüne Seite.

Zuletzt fügte ich in Version 6 noch eine Drehanimation ein. Diese setzte ich bei allen manuellen Inputs, dem Verdrehen und beim Lösen mit `Py222` ein. Der Aufbau ist sehr simpel. Anstelle die Teile einmal mit 90° zu drehen, drehe ich sie 90-mal um 1° und pausiere dazwischen für 1 Millisekunde. So sieht man die Animation.

Der oben genannte Fehler wird durch diese Animation noch witziger, denn die Teile werden nur bis zur türkisfarbenen Mittellinie rotiert. Der Grund dafür ist, dass nach jedem Grad Drehung überprüft wird, ob das Teil links oder rechts der Mittellinie ist. Dies führt zu solch skurrilen Bildern.

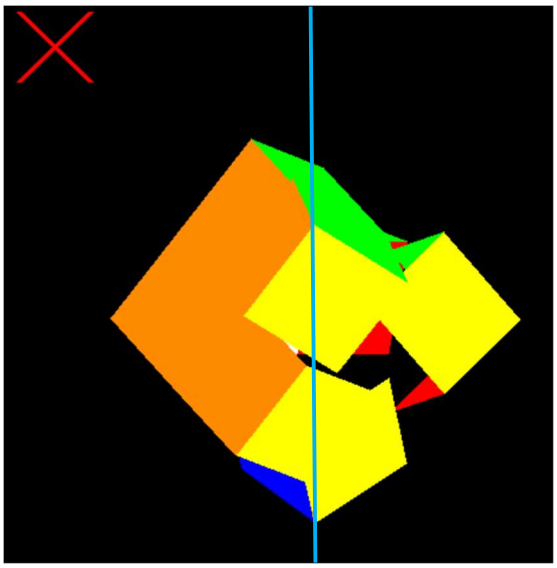


Abb. 19: Cube fehlerhaft nach Animation, Version 6

Aufgrund dieses Fehlers fand ich eine weitere Situation, die den Cube in einen solchen Zustand bringt. Nämlich wenn ein Eckpunkt während der Animation die Mittellinie überschreitet, hört diese Ecke auf zu drehen, da sie nicht mehr auf der rechten Seite ist!

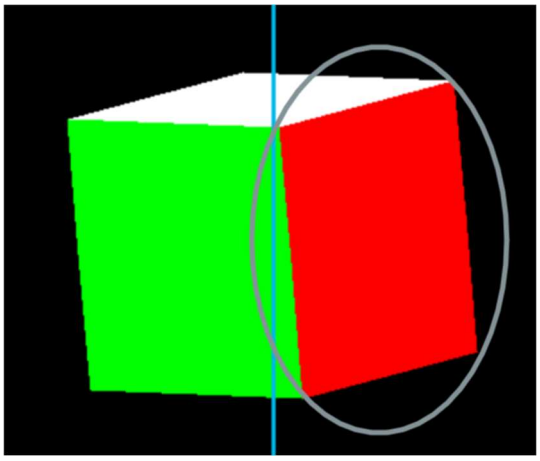


Abb. 21: Cube vor animierter Drehung, Version 6

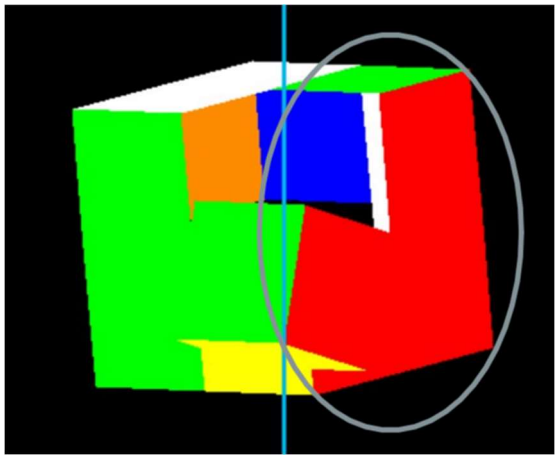


Abb. 20: Cube nach Rotation, Version 6

Hier stoppte die gelb-grün-rote Ecke, als der äusserste Punkt `vec` die türkisfarbene Mittellinie berührte. Das passierte exakt auf der Schnittstelle mit der grauen Aussenlinie der Drehung. Dieser graue Kreis zeigt, wo die Eckpunkte sich bewegen. Man sieht, dass sie sich über die Mittellinie bewegen würden. An der Schnittstelle mit der Mittellinie dreht die gelb-grün-rote Ecke nicht weiter. Vermeidbar ist dieser Fehler immer noch, denn er fiel mir nur theoretisch auf, als ich eine solche Situation simulierte. Beim Lösen des Cube im Programm ist es mir aber nie passiert.

Jetzt war es an der Zeit, die verschiedenen Algorithmen zu testen und die Statistik darüber zu erfassen. Ich liess meinen Laptop mehrere Nächte mit dem Zufallsalgorithmus laufen und sammelte in fast 90 h Laufzeit 50 gelöste Cubes. Dies präsentierte ich Herrn Schaffner und er meinte, ich bräuchte hunderte Zeiten. Das würde mit diesem Programm hunderte Stunden dauern. Deshalb schlug er mir vor, die Display-Updates wegzulassen, damit der Computer nicht unnötige Grafikupdates berechnen muss.

4.8. Version 7

Ziel von Version 7 ist, dass die Prozesse so schnell wie möglich arbeiten können. Ich löschte den gesamten `Py222` Teil, denn ich wollte dieses Programm auch auf anderen Geräten laufen lassen, ohne die zwei Extradokumente mitschicken zu müssen. Danach entfernte ich alle Grafikupdates und Animationen aus dem Verdrehen und Lösen. Ich fügte aber in beide Algorithmen die Leertaste als einmaliges Display-Update ein. So kann ich jederzeit nachschauen, wie weit der Cube bereits ist und ob das Programm noch läuft oder nicht.

Die Effizienz verfünffachte sich auf meinem Rechner und auf einem weiteren Gerät verzehnfachte sie sich. Ich liess Version 7 auf einem zweiten Laptop über eine Woche durchgehend laufen, um genügend Daten für eine Statistik und Auswertung zu sammeln.

5. Auswertung

5.1. Sammeln und Auswerten der Daten

Damit ich möglichst viele Daten sammeln konnte, liess ich meine beiden Zufallsalgorithmen über mehrere Wochen verteilt auf verschiedenen Geräten laufen. 50 Laufzeitstunden später wurde ich von meinem Betreuer darauf hingewiesen, dass ich alle Grafikupdates während dem Lösen weglassen könne. Damit würde sich die Effektivität verbessern, weil die Rechenleistung einzig fürs Drehen gebraucht wird und nicht für die Grafik. Ich tat dies in Version 7 und tatsächlich steigerte sich die Effizienz um das 10-Fache! Es folgt eine kurze Übersicht über die Daten:

	Zufallsalgorithmus	Optimierter Zufallsalgorithmus	Total
Anzahl Solves	804	902	1’706
Anzahl Drehungen ⌀	4’333’588	850’977	2’492’254
Drehungen Total	3’484’204’683	767’581’003	4’251’785’686
Zeit ⌀	20 min 44 sec	4 min 28 sec	12 min 8 sec
Zeit Total	11 Tage 13 h 57 min	2 Tage 19 h 25 min	14 Tage 9h 23 min
Drehungen pro Sekunde ⌀	5’013	4’502	4’743

Tabelle 2: Übersicht gesamt

5.1.1. Auswertung Zufallsalgorithmus

Der Zufallsalgorithmus löste innerhalb von ungefähr 11 Tagen und 14 Stunden 804-mal den 2x2-Zauberwürfel. 50 dieser Solves sind noch mit Grafikupdate, weshalb die Zeiten bei einer Wiederholung der 800 Solves abweichen könnten. Aus diesen Daten berechnet Excel folgende Werte:

Funktion	Drehungen	Zeit
Mittelwert	4’333’588	20 min 44 sec
Median	2’924’12	10 min 54 sec
Minimum	4’798	0.81 sec
Maximum	39’711’617	9 h 7 min 26 sec
Standardabweichung	4’484’435.35	39 min 22 sec
Mittelabweichung	3’235’786.2	18 min 21 sec

Tabelle 3: Übersichtstabelle Zufallsalgorithmus

Da die benötigte Zeit nicht nur von der Anzahl Drehungen, sondern auch vom Gerät abhängt, fokussierte ich mich in der Analyse vor allem auf die Anzahl Drehungen. Die Zeit hängt von der Leistung des Rechners, sonstiger Auslastung des Geräts, der verwendeten Version mit oder ohne Grafikupdate, und natürlich von der Anzahl Drehungen selbst ab. Es ist auch spannend, dass die maximale Zeit nicht den gleichen Solve beschreibt wie die maximale Anzahl Drehungen. Dies, da für den Extremwert der Zeit noch das Programm mit Grafikupdate verwendet wurde. Für die 39 Mio. Drehungen brauchte das Programm nur 4 h. Ich fügte die Daten bezüglich Anzahl Drehungen in eine Grafik ein und machte ein paar spannende Beobachtungen.

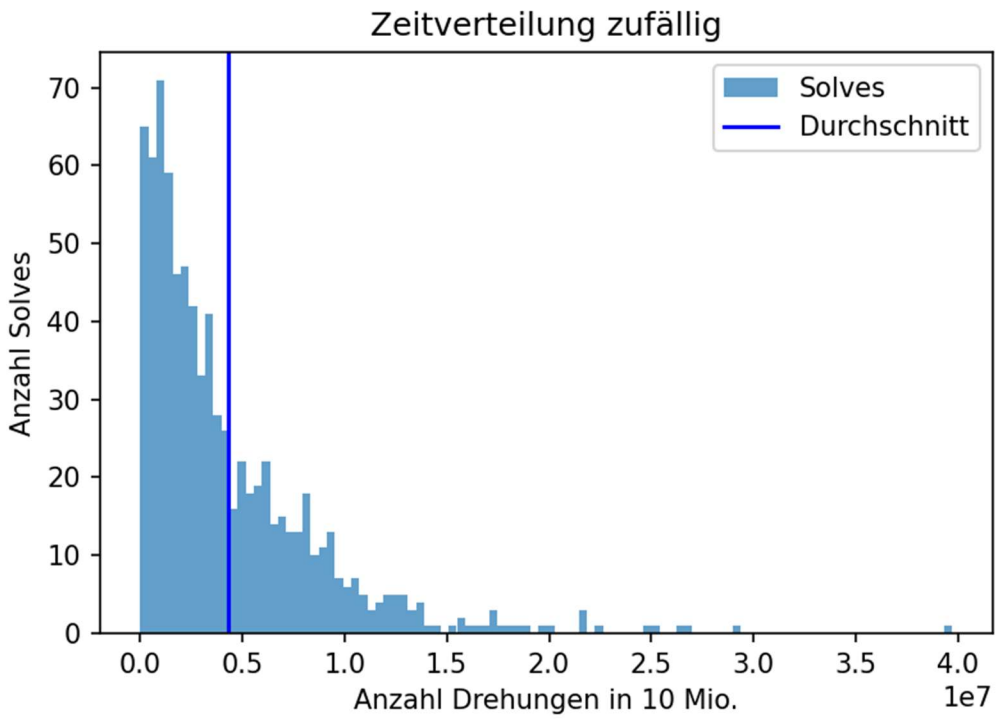


Abb. 22: Grafik Zeitverteilung zufällig

Auf Abb. 22 sieht man, dass der Durchschnitt im unteren Achtel der Grafik liegt. Der höchste Ausschlag liegt im Bereich von 1-2 Millionen Drehungen. Diese kompensieren die Ausreisser nach oben, weshalb der Durchschnitt so tief ist. Weiter erklärt diese Verteilung auch, weshalb sich Standardabweichung und Mittelabweichung so stark unterscheiden. Die Standardabweichung ist um 1,2 Mio. Drehungen grösser, da sie empfindlicher auf Extremwerte reagiert. Die Formel der Mittelabweichung ignoriert die Richtung der Abweichung und kompensiert somit die Extremwerte. Insgesamt liegen 516 Solves unter dem Durchschnitt und 288 Solves darüber. Dies zeigt erneut, dass die Extremwerte einen sehr starken Einfluss auf den Durchschnitt haben. Die untersten **114** Werte zusammen sind kleiner als der grösste Wert.

5.1.2. Auswertung optimierter Zufallsalgorithmus

Auch den optimierten Zufallsalgorithmus liess ich während 2 Tagen und 19 Stunden 902-mal den 2x2-Zauberwürfel lösen. Aus diesem Datenmaterial berechnete ich verschiedene aussagekräftige Werte.

Funktion	Drehungen	Zeit
Mittelwert	850’997	4 min 28 sec
Median	607’784	2 min 28 sec
Minimum	1’101	0.2 sec
Maximum	6’780’981	2 h 55 min
Standardabweichung	835'835.95	8 min 46 sec
Mittelabweichung	620113	3 min 59 sec

Tabelle 4: Übersichtstabelle optimiert

Hier ist wieder der Maximalwert der Zeit nicht gleichzeitig mit der maximalen Anzahl Drehungen aufgezeichnet worden. 6,78 Mio. Drehungen dauerten bloss wenig länger als 20 min, während in den 2 h 55 min nur 1,2 Mio. Drehungen ausgeführt wurden. Zurückzuführen ist dieser Unterschied erneut auf das nicht gelöschte Grafikupdate.

Auch beim optimierten Zufallsalgorithmus überraschte mich die grosse Differenz zwischen Standard- und Mittelabweichung. Diese ist auch hier auf einzelne Extremwerte zurückzuführen, die die Statistik sehr stark beeinflussen. Ich löschte den höchsten Wert, und die Standardabweichung sank um 2 Minuten, während die Mittelabweichung nur um 10 Sekunden sank. Für diesen Algorithmus erstellte ich ebenfalls eine übersichtliche Grafik.

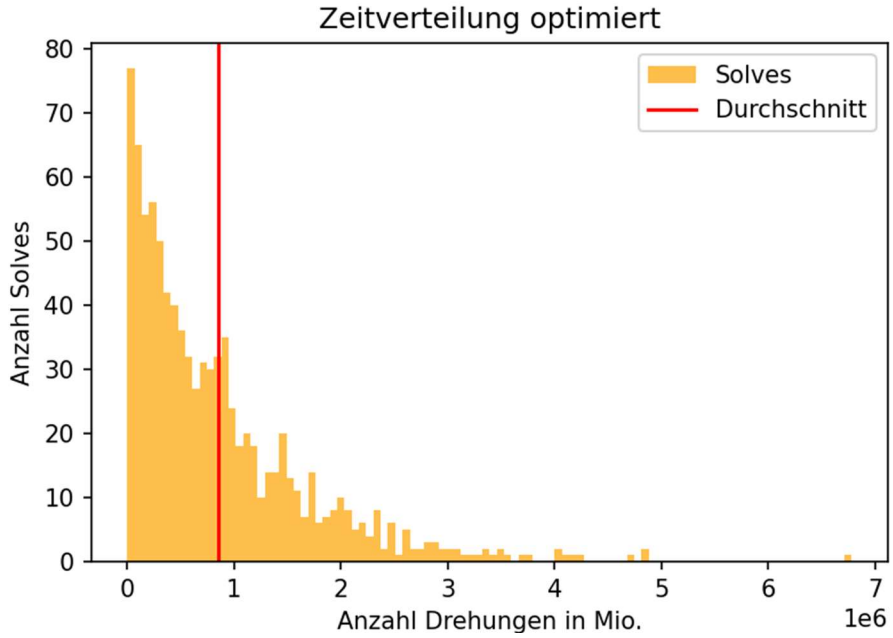


Abb. 23: Grafik Zeitverteilung optimiert

Abb. 23 zeigt genau wie die Abb. 22, dass die Extremwerte stark ins Gewicht fallen. Die Grafiken sehen sehr ähnlich aus, nur in einem fast 10-fach kleineren Zahlenbereich. Der Durchschnitt ist wiederum im unteren Siebtel. Je mehr Drehungen gebraucht wurden, desto weniger Solves gab es, die viele Drehungen benötigten. Bei diesem Algorithmus liegen 345 Solves über dem Durchschnitt und 557 Solves unter dem Durchschnitt. Die untersten **123** Werte zusammengerechnet sind wiederum kleiner als der grösste Wert.

5.1.3. Vergleich Zufall und Optimierung

Nun ist es interessant zu wissen, wie stark sich diese beiden Algorithmen unterscheiden. Dazu fügte ich alle Daten in eine Grafik ein.

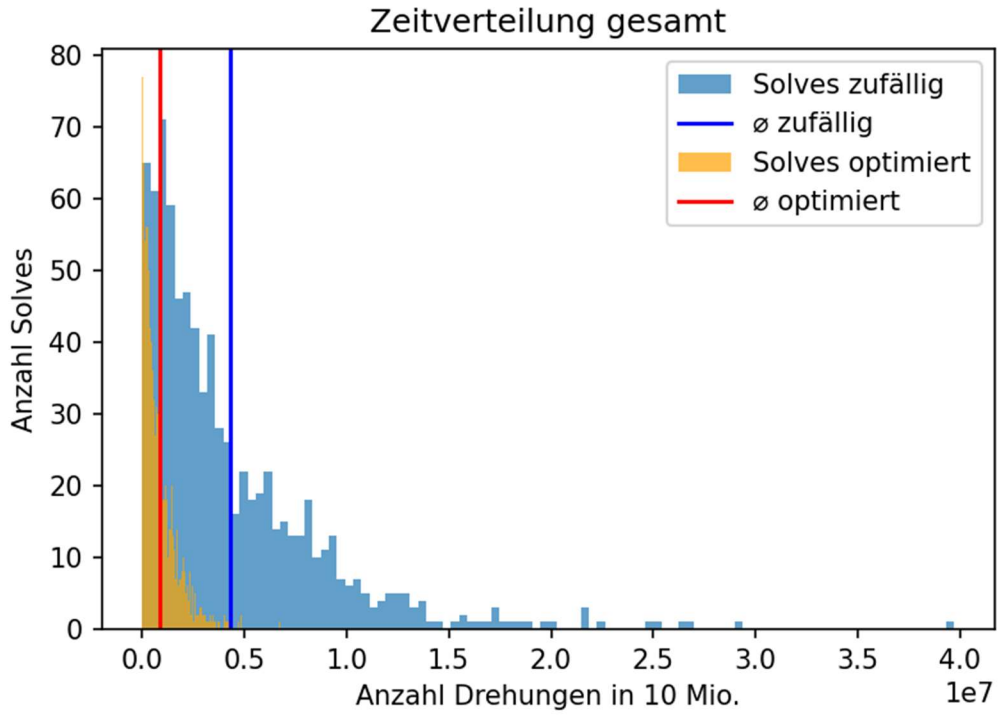


Abb. 24: Zeitverteilung gesamt

Auf dieser Grafik (Abb. 24) ist ersichtlich, dass der optimierte Zufallsalgorithmus um einiges effizienter ist als der reine Zufallsalgorithmus. Diese These wird auch von den Zahlen belegt.

Funktion	Zufall	optimiert
Anzahl Drehungen ⌀	4’333’588	850’977
Zeit ⌀	20 min 44 sec	4 min 28 sec
Drehungen pro Sekunde ⌀	5’013	4’502
Maximalwert Drehungen	39’711’617	6’780’981
Minimalwert Drehungen	4’798	1’101

Tabelle 5: Tabelle Zufall vs. optimiert

In Tabelle 5 ist der bessere Wert grün und der schlechtere Wert rot markiert. Es wird schnell ersichtlich, dass der optimierte Algorithmus in 4 von 5 Disziplinen den besseren Wert aufweist. Die einzige Disziplin, in der der Zufallsalgorithmus besser dasteht, ist die Drehgeschwindigkeit. Diese berechnet sich durch die Anzahl Drehungen, geteilt durch die Zeit in Sekunden. Die Begründung dafür ist einfach. Der optimierte Zufallsalgorithmus führt nach jeder Drehung nämlich viel mehr zeitaufwändige Berechnungen durch als der Zufallsalgorithmus. Ansonsten ist der optimierte Zufallsalgorithmus in jeder Kategorie bei weitem besser. Bei ihm gibt es nur 4 Werte, die grösser sind als der Durchschnitt des Zufallsalgorithmus. Dies zeigt, dass die zwei einfachen Verbesserungen im optimierten Algorithmus in Version 6 und Version 7 einen bemerkenswerten Unterschied bewirkt haben.

5.1.4. Auswertung Py222

Den Solver von Py222 wertete ich auch aus. Es zeigte sich jedoch sehr schnell, dass es höchst unpraktisch ist, viele Daten zu sammeln. Py222 ist mit meinen Ergänzungen dazu ausgelegt, nur einen einzigen Zauberwürfel zu lösen. Danach funktioniert es nicht mehr. Ich verstehe das Programm zu wenig, um es passend zu modifizieren. Aus diesem Grund greife ich auf eine Tabelle aus dem englischen Wikipedia-Artikel[13] zum 2x2-Zauberwürfel zurück.

Anzahl Drehungen	Anzahl Möglichkeiten	Prozentsatz
1	1	0.000027%
2	9	0.00024%
3	54	0.0015%
4	321	0.0087%
5	1’847	0.05%
6	9’992	0.27%
7	50’136	1.36%
8	227’536	6.19%
9	870’072	23.68%
10	1'887’748	51.38%
11	623’800	16.98%
12	2’644	0.072%

Tabelle 6: Kombinationsverteilung Wikipedia

Die Anzahl Drehungen sind hier in 90° und 180° Drehungen angegeben. Dies bedeutet, dass jede mögliche Kombination am 2x2-Zauberwürfel in 12 oder weniger Drehungen lösbar ist und

[13] https://en.wikipedia.org/wiki/Pocket_Cube#Permutations

dass Py222 einen oder sogar mehrere solcher Algorithmen findet. Über die Hälfte aller Kombinationen ist in 9 Drehungen lösbar, weshalb Py222 der effizienteste Lösungsweg ist. Da ich diesen jedoch nur kopiert habe, bewerte ich ihn nicht gleich wie meine eigenen Algorithmen.

5.2. Auswertung Programmierarbeit

Aus meiner Programmierarbeit kann ich rückblickend mehrere Aspekte hervorheben. Als Erstes hat es mir sehr viel Freude bereitet, dieses Projekt zu verwirklichen. Das Programmieren hat mich vor viele Herausforderungen gestellt, die ich meistern musste. Es gab nicht nur Syntax- und Typfehler, sondern auch logische Probleme, bei denen ich nicht wusste und zum Teil immer noch nicht weiss, was die logisch korrekte Lösung ist, die ich programmieren sollte. So hat mein Programm immer noch Fehler in der Drehlogik, die ich nicht lösen kann. Aber es muss im Rahmen dieser Arbeit nicht für jedes Problem eine Lösung gefunden werden. Bei Syntax- und Typfehlern wurde mein Problemlösungsverhalten stark gefordert. Zum Abschluss kann ich sagen, dass es sinnvoll ist, ein solch grosses Projekt klein zu starten, dann mehr und mehr zu erweitern, zu verbessern und komplexere Systeme zu implementieren.

6. Zusammenfassung

In meiner Maturaarbeit programmierte ich in Python-Pygame mithilfe mehrerer Python-Bibliotheken einen 3D-Zauberwürfel, welcher gezeichnet wird, gedreht und gelöst werden kann. Das 3D-Model des Zauberwürfels kann so gedreht werden, dass man ihn aus allen Perspektiven sehen kann. Lösen kann man ihn entweder manuell oder mit einem von drei Lösungsalgorithmen. Der erste Lösungsalgorithmus führt zufällige Drehungen aus, bis der Zauberwürfel gelöst ist. Der zweite Lösungsalgorithmus führt zufällige Drehungen aus, die sich von der vorherigen Drehung unterscheiden müssen. Dies geschieht so lange, bis der Zauberwürfel eine Drehung vom gelösten Zustand entfernt ist. Diese letzte Drehung wird immer gefunden. Der dritte Lösungsalgorithmus ist aus dem Internet heruntergeladen und so angepasst, dass er in meinen Code passt. Dieser dritte Algorithmus findet immer den kürzesten Weg, um den Cube zu lösen.

Diese drei Algorithmen sind unterschiedlich effizient. Der Effizienteste ist der Algorithmus aus dem Internet, welcher ungefähr 83 % aller möglichen Kombinationen des 2x2-Zauberwürfels in 9 oder weniger Drehungen löst. Von meinen zwei selbstgeschriebenen Algorithmen ist der optimierte Zufallsalgorithmus mit durchschnittlich 850'000 Drehungen um durchschnittlich 3.5 Millionen Drehungen effizienter als der Zufallsalgorithmus.

7. Selbstreflexion

Ich hatte sehr viel Freude am Programmieren. Ich bin auch mit dem Endprodukt sehr zufrieden. Jedoch unterschätzte ich, wie viel Zeit ich benötige, um das Modell des Würfels zu erstellen. Ursprünglich rechnete ich damit, in 2 bis 3 Monaten ein Modell erstellt zu haben, damit ich danach einen menschlichen Lösungsalgorithmus programmieren kann. Schlussendlich brauchte ich 7 Monate, um das Modell zu programmieren. Mit noch 1.5 Monaten bis zur Abgabe setzte ich den Fokus mehr auf die schriftliche Arbeit, anstatt auf eine Erweiterung des Codes. Deshalb programmierte ich nach den Zufallsalgorithmen keinen weiteren Algorithmus. Einen solchen Algorithmus in meiner Freizeit zu programmieren, schliesse ich jedoch nicht aus. Dieser wäre dann auf GitHub zu finden. Im Rahmen meiner Maturaarbeit habe ich gelernt, wie man Pygame benutzt, 3D-Modelle erstellt, mit Numpy-Arrays umgeht und wie kleine Verbesserungen einen grossen Einfluss haben können.

8. Quellenangaben

8.1. Quellenverzeichnis

A., Y. (27. Mai 2021). *Youtube*. Abgerufen am 20. Juni 2023 von 3D Rotation & Projection using Python / Pygame: https://www.youtube.com/watch?v=sQDFydEtBLE

Asis, K. D. (19. Juli 2017). *GitHub*. Abgerufen am 29. August 2023 von MeepMoop Py222: https://github.com/MeepMoop/py222

Fincher, J. (05. Oktober 2015). *PyGame: A Primer on Game Programming in Python*. (J. Fincher, Herausgeber, & J. Fincher, Produzent) Abgerufen am 25. Januar 2023 von Real Python: https://realpython.com/pygame-a-primer/

Garron, L. (kein Datum). *alg.cubing.net*. Von 2x2: https://alg.cubing.net/?puzzle=2x2x2

mattwojo. (10. April 2023). *Microsoft Learn*. (Microsoft, Herausgeber) Abgerufen am 15. September 2023 von Erste Schritte bei der Verwendung von Python unter Windows für Anfänger: https://learn.microsoft.com/de-de/windows/python/beginners

o.V. (01. Januar 2000). *Spiel des Jahres*. (S. d. Jahres, Herausgeber) Abgerufen am 22. September 2023 von Rubik's Cube: https://www.spiel-des-jahres.de/spiele/rubiks-cube/

o.V. (15. Oktober 2023). *Wikipedia*. Von Pocket Cube; Permutations: https://en.wikipedia.org/wiki/Pocket_Cube#Permutations

o.V. (01. August 2023). *WorldCubeAssociation*. Von Regulations; Notation: https://www.worldcubeassociation.org/regulations/#article-12-notation

o.V. (kein Datum). *Numpy*. Von Numpy install: https://numpy.org/install/

Shinners, P. (kein Datum). *Pygame*. Von Pygame - Getting started: https://www.pygame.org/wiki/GettingStarted

Wäspi, J. (21. März 2023). *MA_2x2Cube*. Von GitHub: https://github.com/No0ne155/MA_2x2Cube

Werneck, T. (1981). *Der Zauberwürfel.* München: Wilhelm Heyne Verlag.

8.2. Abbildungsverzeichnis

8.3. Tabellenverzeichnis

8.4. Redlichkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Maturaarbeit eigenständig und ohne unerlaubte fremde Hilfe erstellt habe und dass alle Quellen, Hilfsmittel und Internetseiten wahrheitsgetreu verwendet wurden und belegt sind.

Worben, 18.10.2023 J. Wäspi

9. Vollständiger Code, MA_2x2Cube, Version 6

```
# Imports
import pygame
import time
import numpy as np
import py222
import solver
from math import*
from random import*
from pygame import font

# Pygame Initialisieren
pygame.init()
# colors
ORANGE  = ( 255, 140, 0)
ROT     = ( 255, 0, 0)
GRUEN   = ( 0, 255, 0)
SCHWARZ = ( 0, 0, 0)
WEISS   = ( 255, 255, 255)
GELB    = (255,255,0)
BLAU    = (0,0,255)
CLR = (0, 255,255)
WINDOW_SIZE =  600

# Variablen
scale = 100
window = pygame.display.set_mode( (WINDOW_SIZE, WINDOW_SIZE) )
clock = pygame.time.Clock()
center = [(WINDOW_SIZE/2),(WINDOW_SIZE/2)]
state = True
shift = False
turns = ['u','d','f','b','r','l']
font_path = pygame.font.get_default_font()
myfont = pygame.font.Font(font_path, 26)
file_path = 'cubedata.txt'
file_path2 = 'cubedata2.txt'
file_path3 = 'cubedata3.txt'
loop = False
agl = 10
scramblelst = ""
cube222 = py222.initState()
add = ''
oma = False

# Mittelpunkte
xp = np.array([ 1.0,0.0,0.0])
xn = np.array([-1.0,0.0,0.0])
yp = np.array([0.0, 1.0,0.0])
yn = np.array([0.0,-1.0,0.0])
zp = np.array([0.0,0.0, 1.0])
zn = np.array([0.0,0.0,-1.0])

# Klasse Cube
class Cube:
    # Initialisiere Cube mit allen Variabeln
    def __init__(self,vec, vec1, vec2, vec3, p4,p5,p6, coX, coY, coZ, nr) -> None:
        # Vector von Mitte zu Ecke
        self.vec = vec
        # Vektoren von Ecke zu Kante
        self.vecx = vec1 + vec
        self.vecy = vec2 + vec
        self.vecz = vec3 + vec
        # Vektoren von Mitte zu Mitte
        self.p4 = p4
        self.p5 = p5
        self.p6 = p6
        # Farben für x, y und z Seite
        self.coX = coX
        self.coY = coY
        self.coZ = coZ

        self.nr = nr

    # Rotationsalgorithmus für cube-rotation
    def rotate(self, angle, axis):
            rad = angle * pi / 180.0
            c = cos(rad)
            s = sin(rad)
            x = self.vec[0]
            y = self.vec[1]
            z = self.vec[2]
            xx = self.vecx[0]
```

```
            xy = self.vecx[1]
            xz = self.vecx[2]
            yx = self.vecy[0]
            yy = self.vecy[1]
            yz = self.vecy[2]
            zx = self.vecz[0]
            zy = self.vecz[1]
            zz = self.vecz[2]
            p4x = self.p4[0]
            p4y = self.p4[1]
            p4z = self.p4[2]
            p5x = self.p5[0]
            p5y = self.p5[1]
            p5z = self.p5[2]
            p6x = self.p6[0]
            p6y = self.p6[1]
            p6z = self.p6[2]
            if axis == 'x':
                self.vec[1] = y * c - z * s
                self.vec[2] = y * s + z * c
                self.vecx[1] = xy * c - xz * s
                self.vecx[2] = xy * s + xz * c
                self.vecy[1] = yy * c - yz * s
                self.vecy[2] = yy * s + yz * c
                self.vecz[1] = zy * c - zz * s
                self.vecz[2] = zy * s + zz * c
                self.p4[1] = p4y * c - p4z * s
                self.p4[2] = p4y * s + p4z * c
                self.p5[1] = p5y * c - p5z * s
                self.p5[2] = p5y * s + p5z * c
                self.p6[1] = p6y * c - p6z * s
                self.p6[2] = p6y * s + p6z * c
            elif axis == 'y':
                self.vec[0] = x * c + z * s
                self.vec[2] = -x * s + z * c
                self.vecx[0] = xx * c + xz * s
                self.vecx[2] = -xx * s + xz * c
                self.vecy[0] = yx * c + yz * s
                self.vecy[2] = -yx * s + yz * c
                self.vecz[0] = zx * c + zz * s
                self.vecz[2] = -zx * s + zz * c
                self.p4[0] = p4x * c + p4z * s
                self.p4[2] = -p4x * s + p4z * c
                self.p5[0] = p5x * c + p5z * s
                self.p5[2] = -p5x * s + p5z * c
                self.p6[0] = p6x * c + p6z * s
                self.p6[2] = -p6x * s + p6z * c

    # Farben einfüllen und z-Buffer
    def fill(self):
        sor = []
        sor.append([self.p4[2],4])
        sor.append([self.p5[2],5])
        sor.append([self.p6[2],6])
        solt = sorted(sor, key=lambda x: x[0], reverse=True)
        if solt[0][1]==4:
            pygame.draw.polygon(window, self.coY, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecy[0]*100+300, self.vecy[1]*100+300),(self.p4[0]*100+300, self.p4[1]*100+300), (self.vecz[0]*100+300,
self.vecz[1]*100+300)])
            if solt[1][1]==5:
                pygame.draw.polygon(window, self.coZ, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecz[0]*100+300, self.vecz[1]*100+300),(self.p5[0]*100+300, self.p5[1]*100+300), (self.vecx[0]*100+300,
self.vecx[1]*100+300)])
                pygame.draw.polygon(window, self.coX, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecx[0]*100+300, self.vecx[1]*100+300),(self.p6[0]*100+300, self.p6[1]*100+300), (self.vecy[0]*100+300,
self.vecy[1]*100+300)])
            elif solt[1][1]==6:
                pygame.draw.polygon(window, self.coX, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecx[0]*100+300, self.vecx[1]*100+300),(self.p6[0]*100+300, self.p6[1]*100+300), (self.vecy[0]*100+300,
self.vecy[1]*100+300)])
                pygame.draw.polygon(window, self.coZ, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecz[0]*100+300, self.vecz[1]*100+300),(self.p5[0]*100+300, self.p5[1]*100+300), (self.vecx[0]*100+300,
self.vecx[1]*100+300)])
        elif solt[0][1]==5:
            pygame.draw.polygon(window, self.coZ, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecz[0]*100+300, self.vecz[1]*100+300),(self.p5[0]*100+300, self.p5[1]*100+300), (self.vecx[0]*100+300,
self.vecx[1]*100+300)])
            if solt[1][1]==4:
                pygame.draw.polygon(window, self.coY, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecy[0]*100+300, self.vecy[1]*100+300),(self.p4[0]*100+300, self.p4[1]*100+300), (self.vecz[0]*100+300,
self.vecz[1]*100+300)])
                pygame.draw.polygon(window, self.coX, [(self.vec[0]*100+300, self.vec[1]*100+300),
(self.vecx[0]*100+300, self.vecx[1]*100+300),(self.p6[0]*100+300, self.p6[1]*100+300), (self.vecy[0]*100+300,
self.vecy[1]*100+300)])
            elif solt[1][1]==6:
```

```
151.                 pygame.draw.polygon(window, self.coX, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecx[0]*100+300, self.vecx[1]*100+300),(self.p6[0]*100+300, self.p6[1]*100+300), (self.vecy[0]*100+300,
    self.vecy[1]*100+300)])
152.                 pygame.draw.polygon(window, self.coY, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecy[0]*100+300, self.vecy[1]*100+300),(self.p4[0]*100+300, self.p4[1]*100+300), (self.vecz[0]*100+300,
    self.vecz[1]*100+300)])
153.
154.         elif solt[0][1]==6:
155.             pygame.draw.polygon(window, self.coX, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecx[0]*100+300, self.vecx[1]*100+300),(self.p6[0]*100+300, self.p6[1]*100+300), (self.vecy[0]*100+300,
    self.vecy[1]*100+300)])
156.             if solt[1][1]==4:
157.                 pygame.draw.polygon(window, self.coY, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecy[0]*100+300, self.vecy[1]*100+300),(self.p4[0]*100+300, self.p4[1]*100+300), (self.vecz[0]*100+300,
    self.vecz[1]*100+300)])
158.                 pygame.draw.polygon(window, self.coZ, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecz[0]*100+300, self.vecz[1]*100+300),(self.p5[0]*100+300, self.p5[1]*100+300), (self.vecx[0]*100+300,
    self.vecx[1]*100+300)])
159.             elif solt[1][1]==5:
160.                 pygame.draw.polygon(window, self.coZ, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecz[0]*100+300, self.vecz[1]*100+300),(self.p5[0]*100+300, self.p5[1]*100+300), (self.vecx[0]*100+300,
    self.vecx[1]*100+300)])
161.                 pygame.draw.polygon(window, self.coY, [(self.vec[0]*100+300, self.vec[1]*100+300),
    (self.vecy[0]*100+300, self.vecy[1]*100+300),(self.p4[0]*100+300, self.p4[1]*100+300), (self.vecz[0]*100+300,
    self.vecz[1]*100+300)])
162.     # Drehen der Teile um eine Achse
163.     def turn(self, turn, agl):
164.         if turn == 'r':
165.             if self.vec[0] >0:
166.                 lst = [[self.p4[0], 4],[self.p5[0],5],[self.p6[0],6]]
167.                 slst = sorted(lst, key=lambda x: x[0], reverse=True)
168.                 lo = slst[0][1]
169.                 if lo ==4:
170.                     ax = self.p4
171.                 elif lo ==5:
172.                     ax = self.p5
173.                 else:
174.                     ax = self.p6
175.                 self.vec = rot(self.vec,ax,agl)
176.                 self.vecx = rot(self.vecx,ax,agl)
177.                 self.vecy = rot(self.vecy,ax,agl)
178.                 self.vecz = rot(self.vecz,ax,agl)
179.                 self.p4 = rot(self.p4,ax,agl)
180.                 self.p5 = rot(self.p5,ax,agl)
181.                 self.p6 = rot(self.p6,ax,agl)
182.
183.         elif turn == 'l':
184.             if self.vec[0] < 0:
185.                 lst = [[self.p4[0], 4],[self.p5[0],5],[self.p6[0],6]]
186.                 slst = sorted(lst, key=lambda x: x[0])
187.                 lo = slst[0][1]
188.                 if lo ==4:
189.                     ax = self.p4
190.                 elif lo ==5:
191.                     ax = self.p5
192.                 else:
193.                     ax = self.p6
194.                 self.vec = rot(self.vec,ax,agl)
195.                 self.vecx = rot(self.vecx,ax,agl)
196.                 self.vecy = rot(self.vecy,ax,agl)
197.                 self.vecz = rot(self.vecz,ax,agl)
198.                 self.p4 = rot(self.p4,ax,agl)
199.                 self.p5 = rot(self.p5,ax,agl)
200.                 self.p6 = rot(self.p6,ax,agl)
201.
202.         elif turn == 'u':
203.             if self.vec[1] < 0:
204.                 lst = [[self.p4[1], 4],[self.p5[1],5],[self.p6[1],6]]
205.                 slst = sorted(lst, key=lambda x: x[0])
206.                 lo = slst[0][1]
207.                 if lo ==4:
208.                     ax = self.p4
209.                 elif lo ==5:
210.                     ax = self.p5
211.                 else:
212.                     ax = self.p6
213.                 self.vec = rot(self.vec,ax,agl)
214.                 self.vecx = rot(self.vecx,ax,agl)
215.                 self.vecy = rot(self.vecy,ax,agl)
216.                 self.vecz = rot(self.vecz,ax,agl)
217.                 self.p4 = rot(self.p4,ax,agl)
218.                 self.p5 = rot(self.p5,ax,agl)
219.                 self.p6 = rot(self.p6,ax,agl)
220.
221.         elif turn == 'd':
```

```
            if self.vec[1] > 0:
                lst = [[self.p4[1], 4],[self.p5[1],5],[self.p6[1],6]]
                slst = sorted(lst, key=lambda x: x[0], reverse=True)
                lo = slst[0][1]
                if lo ==4:
                    ax = self.p4
                elif lo ==5:
                    ax = self.p5
                else:
                    ax = self.p6
                self.vec = rot(self.vec,ax,agl)
                self.vecx = rot(self.vecx,ax,agl)
                self.vecy = rot(self.vecy,ax,agl)
                self.vecz = rot(self.vecz,ax,agl)
                self.p4 = rot(self.p4,ax,agl)
                self.p5 = rot(self.p5,ax,agl)
                self.p6 = rot(self.p6,ax,agl)

        elif turn == 'b':
            if self.vec[2] > 0:
                lst = [[self.p4[2], 4],[self.p5[2],5],[self.p6[2],6]]
                slst = sorted(lst, key=lambda x: x[0], reverse=True)
                lo = slst[0][1]
                if lo ==4:
                    ax = self.p4
                elif lo ==5:
                    ax = self.p5
                else:
                    ax = self.p6
                self.vec = rot(self.vec,ax,agl)
                self.vecx = rot(self.vecx,ax,agl)
                self.vecy = rot(self.vecy,ax,agl)
                self.vecz = rot(self.vecz,ax,agl)
                self.p4 = rot(self.p4,ax,agl)
                self.p5 = rot(self.p5,ax,agl)
                self.p6 = rot(self.p6,ax,agl)

        elif turn == 'f':
            if self.vec[2] < 0:
                lst = [[self.p4[2], 4],[self.p5[2],5],[self.p6[2],6]]
                slst = sorted(lst, key=lambda x: x[0])
                lo = slst[0][1]
                if lo ==4:
                    ax = self.p4
                elif lo ==5:
                    ax = self.p5
                else:
                    ax = self.p6
                self.vec = rot(self.vec,ax,agl)
                self.vecx = rot(self.vecx,ax,agl)
                self.vecy = rot(self.vecy,ax,agl)
                self.vecz = rot(self.vecz,ax,agl)
                self.p4 = rot(self.p4,ax,agl)
                self.p5 = rot(self.p5,ax,agl)
                self.p6 = rot(self.p6,ax,agl)

    def smootht(self, turn, agl):
        for i in range(agl):
            self.turn(f'{turn}',i)
            pygame.time.wait(10)

# Den 8 Cubes ihre Werte zuweisen
cube1 = Cube(np.array([-1.0,-1.0, 1.0]), np.array([ 1,0,0]), np.array([0, 1,0]), np.array([0,0,-1]), np.copy(xn),
    np.copy(yn), np.copy(zp),BLAU, ORANGE, WEISS,1)
cube2 = Cube(np.array([ 1.0,-1.0, 1.0]), np.array([-1,0,0]), np.array([0, 1,0]), np.array([0,0,-1]), np.copy(xp),
    np.copy(yn), np.copy(zp),BLAU, ROT, WEISS,2)
cube3 = Cube(np.array([ 1.0, 1.0, 1.0]), np.array([-1,0,0]), np.array([0,-1,0]), np.array([0,0,-1]), np.copy(xp),
    np.copy(yp), np.copy(zp),BLAU, ROT, GELB,3)
cube4 = Cube(np.array([-1.0, 1.0, 1.0]), np.array([ 1,0,0]), np.array([0,-1,0]), np.array([0,0,-1]), np.copy(xn),
    np.copy(yp), np.copy(zp),BLAU, ORANGE, GELB,4)
cube5 = Cube(np.array([-1.0,-1.0,-1.0]), np.array([ 1,0,0]), np.array([0, 1,0]), np.array([0,0, 1]), np.copy(xn),
    np.copy(yn), np.copy(zn),GRUEN, ORANGE, WEISS,5)
cube6 = Cube(np.array([ 1.0,-1.0,-1.0]), np.array([-1,0,0]), np.array([0, 1,0]), np.array([0,0, 1]), np.copy(xp),
    np.copy(yn), np.copy(zn),GRUEN, ROT, WEISS,6)
cube7 = Cube(np.array([ 1.0, 1.0,-1.0]), np.array([-1,0,0]), np.array([0,-1,0]), np.array([0,0, 1]), np.copy(xp),
    np.copy(yp), np.copy(zn),GRUEN, ROT, GELB,7)
cube8 = Cube(np.array([-1.0, 1.0,-1.0]), np.array([ 1,0,0]), np.array([0,-1,0]), np.array([0,0, 1]), np.copy(xn),
    np.copy(yp), np.copy(zn),GRUEN, ORANGE, GELB,8)

# z-Buffer
def buffer():
    sort = []
    sort.append([cube1.vec[2],1])
    sort.append([cube2.vec[2],2])
    sort.append([cube3.vec[2],3])
```

```
    sort.append([cube4.vec[2],4])
    sort.append([cube5.vec[2],5])
    sort.append([cube6.vec[2],6])
    sort.append([cube7.vec[2],7])
    sort.append([cube8.vec[2],8])

    sorted_list = sorted(sort, key=lambda x: x[0], reverse = True)

    for i in range(1, 8):
        cubelet = globals()['cube{}'.format(sorted_list[i][1])]
        cubelet.fill()

# Definion für Rotationsberechnung
def rot(vec, ax,agl):
    v_x = vec[0]
    v_y = vec[1]
    v_z = vec[2]
    c = cos(np.radians(agl))
    s = sin(np.radians(agl))
    r = 1-c
    axe = ax/np.linalg.norm(ax)
    u_x = axe[0]
    u_y = axe[1]
    u_z = axe[2]
    v_rotated_x = v_x * (c + u_x**2 * r) + v_y * (u_x * u_y * r - u_z * s) + v_z * (u_x * u_z * r + u_y * s)
    v_rotated_y = v_x * (u_y * u_x * r + u_z * s) + v_y * (c + u_y ** 2 * r) + v_z * (u_y * u_z * r - u_x * s)
    v_rotated_z = v_x * (u_z * u_x * r - u_y * s) + v_y * (u_z * u_y * r + u_x * s) + v_z * (c + u_z**2 * r)
    newvec = np.array([v_rotated_x, v_rotated_y, v_rotated_z])
    return newvec

#Zeichen ob gelöst oder nicht
def mark(res):
    global state
    if res == 't':
        pygame.draw.line(window, GRUEN, (80,60), (130,10), 5)
        pygame.draw.line(window, GRUEN, (80,60), (50,30), 5)
        state = True
    if res == 'f':
        pygame.draw.line(window, ROT, (50,30), (110,90), 5)
        pygame.draw.line(window, ROT, (110,30), (50,90), 5)
        state = False

# Überprüfung ob alle Teile untereinander an der richtigen Position sind
def checker():
    if np.allclose(cube1.vecx, cube2.vecx, 0.00001) == True:
        if np.allclose(cube5.vecx, cube6.vecx, 0.00001) == True:
            if np.allclose(cube2.vecz, cube6.vecz, 0.00001) == True:
                if np.allclose(cube3.vecx, cube4.vecx, 0.00001) == True:
                    if np.allclose(cube7.vecx, cube8.vecx, 0.00001) == True:
                        if np.allclose(cube4.vecz, cube8.vecz, 0.00001) == True:
                            if np.allclose(cube8.vecy, cube5.vecy, 0.00001) == True:
                                mark('t')
                            else:
                                mark('f')
                        else:
                            mark('f')
                    else:
                        mark('f')
                else:
                    mark('f')
            else:
                mark('f')
        else:
            mark('f')
    else:
        mark('f')

# Zufallsalgorithmus der den Cube löst
def solveR():
    global loop
    count = 0
    t0=time.time()
    run = True
    while state == False and run == True:
        window.fill((0,0,0))
        k = randint(0,5)
        dir = randint(0,1)
        dire = [-90,90]
        turn = turns[k]
        for i in range(1, 9):
            cubelet = globals()['cube{}'.format(i)]
            cubelet.turn(f'{turn}', dire[dir])
        count = count+1
        display_text(f"Moves: {count}", 370, 50)
        if loop == True:
```

```
            display_text('L', 10,10)
        if loop == False:
            display_text('N', 10,10)
        display_text('R', 550,570)
        checker()
        buffer()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print(count)
                exit()
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    print(count)
                    exit()
                elif event.key == pygame.K_0:
                    run = False
                elif event.key == pygame.K_w:
                    if loop == True:
                        loop = False
                    elif loop == False:
                        loop = True
        pygame.display.update()
    print(count)
    t1 = time.time()
    ts = t1-t0
    tm = ts/60
    th = tm/60
    print(f'Your Time was{ts} seconds.')
    print(f'This is {tm} minutes')
    print(f'Or {th} hours!!')
    print(f'Solved is {state}')
    if state:
        with open(file_path, 'a') as file:
            file.write(f'Finished random in: {ts} sec, {count} turns.'+'\n')

# Überprüfung ob "one move away"
def omaCheck():
    global oma

    oma = False
    if np.allclose(cube1.vecx, cube2.vecx, 0.00001) == True:
        if np.allclose(cube5.vecx, cube6.vecx, 0.00001) == True:
            if np.allclose(cube2.vecz, cube6.vecz, 0.00001) == True:
                if np.allclose(cube3.vecx, cube4.vecx, 0.00001) == True:
                    if np.allclose(cube7.vecx, cube8.vecx, 0.00001) == True:
                        if np.allclose(cube4.vecz, cube8.vecz, 0.00001) == True:
                            oma = True
    elif np.allclose(cube1.vecx, cube2.vecx, 0.00001) == True:
        if np.allclose(cube3.vecx, cube4.vecx, 0.00001) == True:
            if np.allclose(cube1.vecy, cube4.vecy, 0.00001) == True:
                if np.allclose(cube5.vecx, cube6.vecx, 0.00001) == True:
                    if np.allclose(cube7.vecx, cube8.vecx, 0.00001) == True:
                        if np.allclose(cube5.vecy, cube8.vecy, 0.00001) == True:
                            oma = True
    elif np.allclose(cube1.vecy, cube4.vecy, 0.00001) == True:
        if np.allclose(cube5.vecy, cube8.vecy, 0.00001) == True:
            if np.allclose(cube5.vecz, cube1.vecz, 0.00001) == True:
                if np.allclose(cube2.vecy, cube3.vecy, 0.00001) == True:
                    if np.allclose(cube6.vecy, cube7.vecy, 0.00001) == True:
                        if np.allclose(cube2.vecz, cube6.vecz, 0.00001) == True:
                            oma = True

# Optimierter Zufallsalgorithmus
def solveR2():
    global loop
    count = 0
    t0=time.time()
    run = True
    lastTurn = 'x'
    while state == False and run == True:
        window.fill((0,0,0))
        omaCheck()
        if oma == False and state == False:
            dir = randint(0,2)
            dire = [-90, 90, 180]
            if lastTurn == 'x':
                t = ['u','d','f','b','r','l']
                k = randint(0,5)
                turn = t[k]
            elif lastTurn == 'u' or lastTurn == 'd':
                t = ['f','b','r','l']
                k = randint(0,3)
                turn = t[k]
            elif lastTurn == 'f' or lastTurn == 'b':
                t = ['d','u','r','l']
```

```
                k = randint(0,3)
                turn = t[k]
            elif lastTurn == 'r' or lastTurn == 'l':
                t = ['f','b','u','d']
                k = randint(0,3)
                turn = t[k]
            lastTurn=turn
            for i in range(1, 9):
                cubelet = globals()['cube{}'.format(i)]
                cubelet.turn(f'{turn}', dire[dir])
            count = count+1

        elif oma == True:
            for i in range(1, 9):
                cubelet = globals()['cube{}'.format(i)]
                cubelet.turn('u',-90)
            checker()
            if state == False and oma == True:
                for i in range(1, 9):
                    cubelet = globals()['cube{}'.format(i)]
                    cubelet.turn('u',-90)
                checker()
                if state == False and oma == True:
                    for i in range(1, 9):
                        cubelet = globals()['cube{}'.format(i)]
                        cubelet.turn('u',-90)
                    checker()
                    if state == False and oma == True:
                        for i in range(1, 9):
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('u',-90)
                        checker()
                        if state == False and oma == True:
                            for i in range(1, 9):
                                cubelet = globals()['cube{}'.format(i)]
                                cubelet.turn('r',-90)
                            checker()
                            if state == False and oma == True:
                                for i in range(1, 9):
                                    cubelet = globals()['cube{}'.format(i)]
                                    cubelet.turn('r',-90)
                                checker()
                                if state == False and oma == True:
                                    for i in range(1, 9):
                                        cubelet = globals()['cube{}'.format(i)]
                                        cubelet.turn('r',-90)
                                    checker()
                                    if state == False and oma == True:
                                        for i in range(1, 9):
                                            cubelet = globals()['cube{}'.format(i)]
                                            cubelet.turn('r',-90)
                                        checker()
                                        if state == False and oma == True:
                                            for i in range(1, 9):
                                                cubelet = globals()['cube{}'.format(i)]
                                                cubelet.turn('f',-90)
                                            checker()
                                            if state == False and oma == True:
                                                for i in range(1, 9):
                                                    cubelet = globals()['cube{}'.format(i)]
                                                    cubelet.turn('f',-90)
                                                checker()
                                                if state == False and oma == True:
                                                    for i in range(1, 9):
                                                        cubelet = globals()['cube{}'.format(i)]
                                                        cubelet.turn('f',-90)
                                                    checker()
                                                    if state == False and oma == True:
                                                        for i in range(1, 9):
                                                            cubelet = globals()['cube{}'.format(i)]
                                                            cubelet.turn('f',-90)
                                                        checker()
            count = count+1

        display_text(f"Moves: {count}", 370, 50)
        if loop == True:
            display_text('L', 10,10)
        if loop == False:
            display_text('N', 10,10)
        display_text('R2', 550,570)
        checker()
        buffer()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print(count)
```

```
                exit()
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    print(count)
                    exit()
                elif event.key == pygame.K_0:
                    run = False
                elif event.key == pygame.K_w:
                    if loop == True:
                        loop = False
                    elif loop == False:
                        loop = True
        pygame.display.update()
    print(count)
    t1 = time.time()
    ts = t1-t0
    tm = ts/60
    th = tm/60
    print(f'Your Time was{ts} seconds.')
    print(f'This is {tm} minutes')
    print(f'Or {th} hours!!')
    print(f'Solved is {state}')
    if state:
        with open(file_path3, 'a') as file:
            file.write(f'Finished random in: {ts} sec, {count} turns.'+'\n')

# Verdrehalgorithmus
def scramble():
    global scramblelst
    for j in range(25):
        k = randint(0,5)
        turn = turns[k]
        dir = randint(0,2)
        dire = [-90,90,180]
        if dire[dir] == -90:
            add = ''
        elif dire[dir] == 90:
            add = "'"
        elif dire[dir] == 180:
            add = '2'
        scramblelst =scramblelst + turn.upper()+add+' '
        for k in range(30):
            d = dire[dir]
            for i in range(1, 9):
                window.fill((0,0,0))
                cubelet = globals()['cube{}'.format(i)]
                cubelet.turn(f'{turn}',(d/30))
                buffer()
                pygame.display.update()
            pygame.time.wait(1)
    print(scramblelst)

# py222 Implementation
def solve222():
    global cube222
    t0 = time.time()
    cube222 = py222.doAlgStr(cube222, scramblelst)
    alg = solver.solveCube(cube222)
    print(alg)
    algs = solver.solvedalg
    head, sep, tail = algs.partition("gap")
    algs = head.split()
    for i in range(len(algs)):
        algs[i]=algs[i].lower()
    for i in range(len(algs)):
        if len(algs[i]) == 2:
            if algs[i][1] == '2':
                for j in range(90):
                    for k in range(1, 9):
                        window.fill((0,0,0))
                        cubelet = globals()['cube{}'.format(k)]
                        cubelet.turn(f'{algs[i][0]}',2)
                        buffer()
                        pygame.display.update()
                    pygame.time.wait(1)
            elif algs[i][1] == "'":
                for j in range(45):
                    for k in range(1, 9):
                        window.fill((0,0,0))
                        cubelet = globals()['cube{}'.format(k)]
                        cubelet.turn(f'{algs[i][0]}',2)
                        buffer()
                        pygame.display.update()
                    pygame.time.wait(1)
        else:
```

```
            for j in range(45):
                for k in range(1, 9):
                    window.fill((0,0,0))
                    cubelet = globals()['cube{}'.format(k)]
                    cubelet.turn(f'{algs[i][0]}',-2)
                    buffer()
                    pygame.display.update()
                pygame.time.wait(1)
    t1 = time.time()
    ti = t1 - t0
    with open(file_path2, 'a') as file:
        file.write(f'Finished random in: {ti} sec, {len(algs)} turns.'+'\n')
    print(ti, len(algs))

# Wilkommensnachricht
def welcome():
    print('Wilkommen zu meinem 2x2 Cube simulator')
    print('Um zu beginnen, können sie den Cube mit den Pfeiltasten bewegen')
    print('Um eine Drehung vorzunehmen, drücken Sie eine der folgenden Tasten:')
    print('U, D, L, R, F, B')
    print('Dies nimmt eine Rotation der entsprechenden seite in Uhrzeigerrichtung vor')
    print('Um eine Drehung gegen den Uhrzeigersinn vorzunehmen, drücken Sie zusätzlich "Shift Links"')
    print('Um den Cube verdrehen zu lassen, drücken Sie die "1"')
    print('Um den Cube von einem Zufallsgenerator lösen zu lassen, drücken sie die "2"')
    print('Während dem Lösen durch zufalls, können Sie den Prozess mit "0" abbrechen.')
    print('Mit der Taste "W" könnens Sie in den Loop eintreten, d.h. nach dem Lösen, verdreht er sich wieder, und beginnt zu lösen.')
    print('Mit der Taste "3" können Sie einen Effizienteren Zufallsalgorithmus zum Lösen wählen')
    print('Mit der Taste "4" können Sie den Cube mit einem Effizienzsolver lösen.')
    print('Taste 4 bitte nur nach neustarten des Programms und nur dem Verdrehalgorithmus betätigen.')

# Definition die Text am Bildschirm anzeigt
def display_text(text, x, y):
    text_surface = myfont.render(text, True, WEISS)
    window.blit(text_surface, (x, y))

# Cube in Position bringen
def setupR():
    for i in range(1, 9):
        cubelet = globals()['cube{}'.format(i)]
        cubelet.rotate(agl, 'y')
    for i in range(1, 9):
        cubelet = globals()['cube{}'.format(i)]
        cubelet.rotate(agl, 'x')

setupR()
welcome()

# Haupt-schleife
running = True
agl = 5
while running == True:
    clock.tick(60)
    window.fill((0,0,0))

    buffer()
    checker()

    # Tastenabfragen
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                for i in range(1, 9):
                    cubelet = globals()['cube{}'.format(i)]
                    cubelet.rotate(-agl, 'y')
            elif event.key == pygame.K_LEFT:
                for i in range(1, 9):
                    cubelet = globals()['cube{}'.format(i)]
                    cubelet.rotate(agl, 'y')
            elif event.key ==pygame.K_UP:
                for i in range(1, 9):
                    cubelet = globals()['cube{}'.format(i)]
                    cubelet.rotate(-agl, 'x')
            elif event.key == pygame.K_DOWN:
                for i in range(1, 9):
                    cubelet = globals()['cube{}'.format(i)]
                    cubelet.rotate(agl, 'x')
            elif event.key == pygame.K_ESCAPE:
                running = False
            elif event.key == pygame.K_r:
                if not shift:
                    for j in range(90):
                        for i in range(1, 9):
```

```
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('r',-1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)

                if shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('r',1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
            elif event.key == pygame.K_l:
                if not shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('l',-1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
                if shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('l',1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
            elif event.key == pygame.K_u:
                if not shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('u',-1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
                if shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('u',1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
            elif event.key == pygame.K_d:
                if not shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('d',-1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
                if shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('d',1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
            elif event.key == pygame.K_f:
                if not shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('f',-1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
                if shift:
                    for j in range(90):
```

```
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('f',1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
            elif event.key == pygame.K_b:
                if not shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('b',-1)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
                if shift:
                    for j in range(90):
                        for i in range(1, 9):
                            window.fill((0,0,0))
                            cubelet = globals()['cube{}'.format(i)]
                            cubelet.turn('b',90)
                            buffer()
                            pygame.display.update()
                        pygame.time.wait(1)
            elif event.key == pygame.K_a:
                for j in range(45):
                    for i in range(1, 9):
                        window.fill((0,0,0))
                        cubelet = globals()['cube{}'.format(i)]
                        cubelet.turn('r',-1)
                        buffer()
                        pygame.display.update()
                    pygame.time.wait(1)
            elif event.key == pygame.K_1:
                scramble()
            elif event.key == pygame.K_2:
                solveR()
            elif event.key == pygame.K_LSHIFT:
                shift = True
            elif event.key == pygame.K_w:
                if loop:
                    loop = False
                elif not loop:
                    loop = True
            elif event.key == pygame.K_3:
                solveR2()
            elif event.key == pygame.K_4:
                solve222()
        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_LSHIFT:
                shift = False

    if loop:
        scramble()
        solveR()

    pygame.display.update()
```