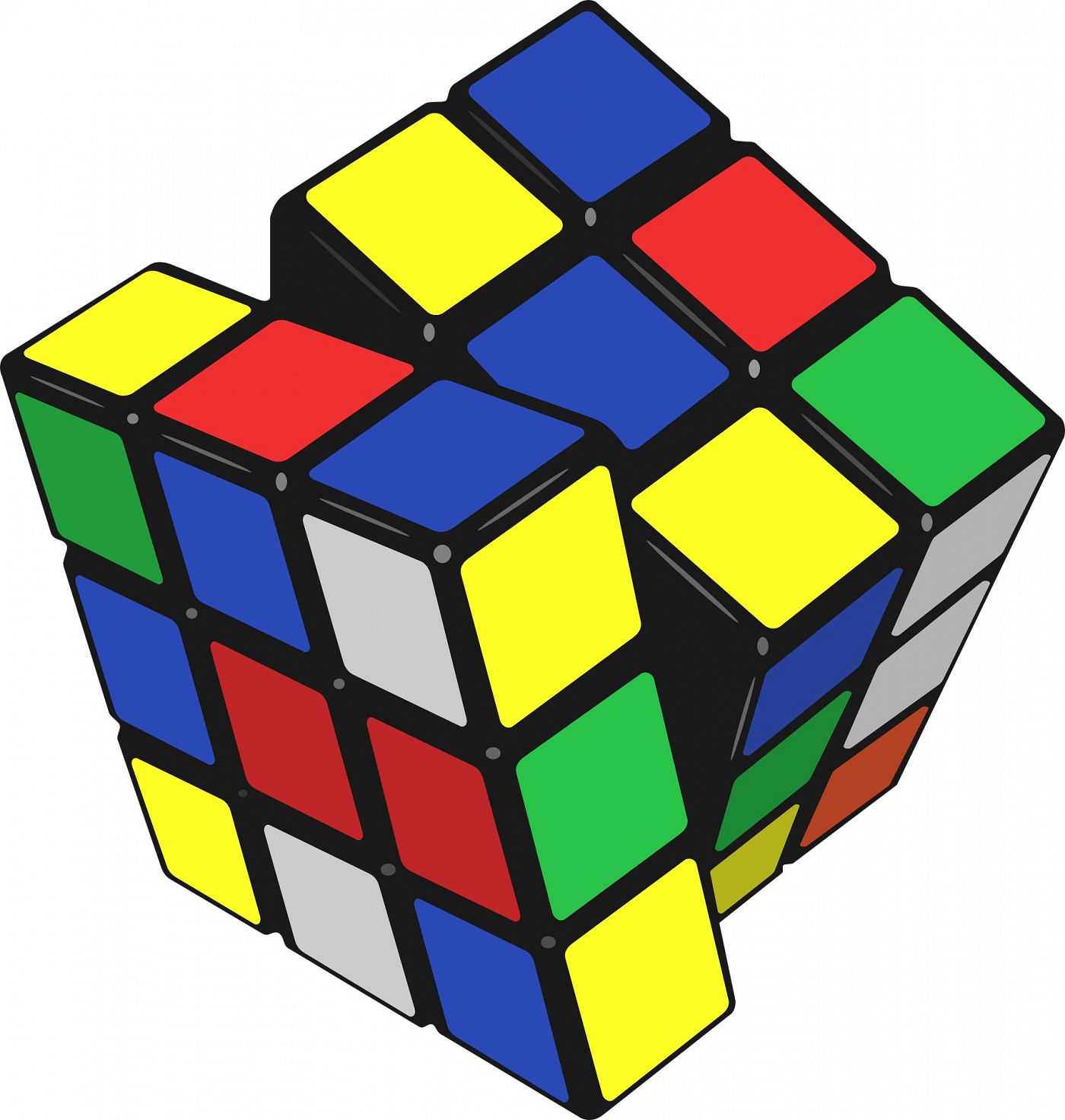
Rubik’s Cube Solver



Figuur 1. Rubik's Cube. Bron: <https://anyfmalifestyle.nl/product/rubik-s-cube/47629>

Naam: Martijn de Vries

Datum: 18-04-2022

Versie: 1.1

Inhoudsopgave

1. **Doel van het project**

Het doel is van dit project is om een Rubik’s Cube solver te maken die mechanisch wordt aangestuurd door een Arduino microcontroller. Een Rubik’s kubus zal zich in een frame bevinden en in dit frame zijn en aantal kleine motoren gebouwd. De motoren bevinden zich aan alle zijden van de kubus zodat alle zes de vlakken geroteerd kunnen, zowel links als rechtsom. Met camera’s zal de configuratie van de kubus bepaald worden. Er zal dus zowel mechanisch als ICT-werk plaatsvinden. Voor nu doe ik het project alleen, maar ik zou het niet erg vinden als, vooral voor het mechanische gedeelte mensen zouden instappen om daaraan mee te werken.

De opzet van het project is om eerst een werkende Rubik’s kubus te modelleren in een 3d engine library in python. Vervolgens zal een code worden geschreven om deze te randomiseren en op te lossen.   
De Arduino microcontroller werkt niet met python code. Dus de code zal daarna moeten worden geconverteerd van python code naar een code waar de Arduino op kan draaien. Dit is gelijk een persoonlijk doel omdat ik nog nooit met een Arduino heb gewerkt.

Tegelijkertijd zal aan de mechanische constructie gebouwd worden zodat de geschreven code voor de Arduino ook getest kan worden.

1. **De kubus modelleren in Ursina**

Ursina is een library voor python die het mogelijk maakt om 3D modellen te maken. De library is eentje die gebouwd is door developers uit de Panda3D library. Dit hebben ze gedaan om de Panda3D library te vereenvoudigen. Aangezien ik toch niet op zoek ben naar veel functies in een library is het eenvoudiger om Ursina te gebruiken. Deze geeft en 3D omgeving waarin objecten geplaatst, verplaatst, geroteerd en kleur gegeven kunnen worden. Dit is voldoende voor het modelleren van een Rubik’s kubus.

Er zal een klasse worden gemaakt die één blokje van de kubus kan maken. Dit is nodig omdat de individuele blokjes van de kubus in kleur allemaal verschillend zijn. Het is mogelijk om 6 zogenaamde “quads” te combineren tot 1 kubus. Deze “quads” zijn 2-dimensionale vlakken met dezelfde hoogte en breedte. Door optioneel te maken welke kleur deze vlakken kan de kubus opgebouwd worden met 27 (3x3x3) verschillende van deze objecten. Deze worden gepositioneerd in de 3D omgeving aan de hand van een voorbereide positie lijst. Deze lijst is een 3-dimensianale array waarin de x-, y- en z-posities worden bijgehouden.

Wanneer een zijde van de kubus geraaid wordt verandert de positie van de 8 objecten (de middelste hoeft niet mee te draaien, die blijft altijd hetzelfde). Aan de hand van de positielijst zal dit gedaan worden. Echter verandert niet alleen de positie van elk blokje maar ook de kleur die zich op dat zich moment aan elke zijde van dat blokje bevindt. Er is in een klein vooronderzoek geprobeerd om de individuele blokken te roteren maar omdat elk object zijn eigen assenconfiguratie relateert aan de wereldoriëntatie leidt dit tot onnodig gecompliceerde berekeningen. Daarom wordt er voor elke rotatie 8 objecten verwijdert en nieuwe aangemaakt waarbij de kleuren per blokje worden geroteerd zodat ze in de nieuwe positie de juiste weergave hebben. Het draaien van de zijden van de kubus zal via key-bindings worden gedaan via een input functie die onderdeel is van de Ursina library.

Verder zal de camera in de 3D omgeving draaibaar zijn. We maken de code zo dat wanneer de camera gericht is een op bepaald vlak dat alleen dit vlak draait wanneer er een input wordt gedaan. Dit maakt het aantal input minder en kunnen we met de cijfers op het num-pad alle rotaties doen.

1. **Solver code in Python**

De oplossing van de kubus zal gedaan worden aan de hand van de zogenaamde laag-voor laag methode. Zie bijlage voor de instructies voor een uitgebreidere versie het oplossen van de kubus volgens deze methode. In het kort zijn dit de stappen die we gaan coderen:

* Het oplossen van de witte randen
* De witte hoeken goed zetten
* De middelste laag oplossen
* Het bovenste kruis in positie brengen
* De bovenste randen oplossen
* Bovenste hoeken in positie brengen
* De bovenste hoeken oplossen

Elk van deze stappen zal in een aparte functie worden gedefinieerd. De volgende functie wordt pas aangeroepen wanneer de vorige functie helemaal is opgelost. Wanneer alle stappen zijn opgelost wordt er een signaal gegeven dat de kubus is opgelost en zullen de rotaties ophouden.

**Oplossen van de witte randen**

Dit is voor de mens de meest eenvoudige stap omdat er bijna nog geen mogelijkheden zijn om de vorige opgeloste stappen weer ongedaan te maken. Echter voor de code is dit het langste code omdat de alle posities waar het blokje zich kan bevinden nog gecheckt moet worden. We doen dit door naar elke rand te kijken (dit zijn er 10) en vervolgens naar de oriëntatie van het witte vlak. Aan de hand hiervan geven we het aantal rotaties terug. Vervolgens wordt opnieuw gekeken naar alle posities. Wanneer alle witte randen zich in het witte vlak bevinden met de juiste oriëntatie zal dit worden teruggegeven en zal de code doorgaan naar de volgende functie Dit is een relatief eenvoudige manier om te coderen maar voor een eerste versie is het voldoende. We kunnen in een later stadium kijken naar andere, meer gecompliceerde algoritmes, maar voor de eerste versie is het doel om de solver werkend te krijgen.

**Oplossen van de witte hoeken**

Gaat eigenlijk op dezelfde manier als het oplossen van de witte randen. Het verschil is dat we nu kijken naar 8 posities waarin zich de hoeken kunnen bevinden. En omdat een hoek 3 kleur vlakken heeft zijn er dus 3 verschillende oriëntaties waarin zich de hoeken kunnen bevinden. Dit vraagt om 1 extra conditionele vergelijking in de code.

**Oplossen van de middelste laag**

In de middelste laag bevinden zich alleen vlakken met twee (of één) kleurvlakken. Daarom hoeven we alleen te kijken naar rand-blokken. We kijken eerst naar alle bovenste randen. Wanneer de het voorste vlak van deze rand dezelfde kleur heeft als de zijde waar hij in hoort dan is er meteen een oplossing mogelijk voor dit blokje. Wanneer hij zich in een andere oriëntatie bevindt (kleur van het vlak waar hij heen moet zit boven) dan moeten we hetzelfde algoritme twee keer uitvoeren (zie bijlage). Wanneer er geen van beide wordt gevonden en de middelste laag is nog niet opgelost (randen bevinden zich op de goede plek maar in de verkeerde oriëntatie) dan wordt het algoritme toegepast op een blokje die niet goed zit, zodat deze wordt verplaatst naar het bovenste vlak. Daarna draaien we het bovenste vlak net zo lang totdat het kleurvlak aan de voorkant overeenkomt met de zijde waar hij naar toe moet en voeren we het algoritme hoort dat pas (links of rechts plaatsen, zie bijlage)

**Bovenste kruis in positie zetten**

Wanneer de middelste laag kan het zijn (in de meeste gevallen) dat alle bovenste blokjes niet juist georiënteerd zijn. We kijken eerst naar de bovenste (gele) randen. Wanneer deze een kruis vormen zijn deze goed georiënteerd, in dat geval zullen er geen algoritmes uitgevoerd worden in deze functie. Wanneer dit niet het geval is zijn er vier verschillende oriëntaties waarin zich de gele randen kunnen bevinden.

* Puntje. Alleen de middelste is goed
* L-vorm (deze kan weer in vier oriëntaties wijzend noord, oost, zuid of west)
* Lijn (horizontaal of verticaal)
* Kruis

We lussen door de blokjes in het bovenste vlak en kijken welke de gele kant al boven heeft zitten. Aan de hand hiervan bepalen we met welke vorm we te maken hebben en zullen per vorm een algoritme uit worden gevoerd. Alleen bij het puntje wordt het twee keer uitgevoerd omdat na 1 keer uitvoeren van het algoritme na het puntje een L-vorm de uitkomst zal zijn.

**Bovenste randen oplossen**

Wanneer alle bovenste randen in de juist oriëntatie zijn gedraaid zullen ze in de juist positie worden gezet. We draaien eerst met een algoritme de groene (voorste vlak).   
Omdat er maar 1 algoritme is die uitgevoerd moet worden, maar hoe vaak afhankelijk is van de positie van de andere 3 randen maken we een lijst met de kleuren van de zijden (anders dan de bovenste gele zijde) met de klokt mee. Aan de hand hiervan kunnen we bepalen, hoe vaak en vanaf welke positie het algoritme uitgevoerd moet worden.

**Bovenste hoeken in positie brengen**

We maken weer een lijst met de posities met de klok mee waarin zich de hoeken bevinden. Er zitten altijd 0, 1 of 4 hoeken in de goede positie. En er is wederom 1 algoritme. We bepalen het aantal hoeken wat goed zit. Als er 0 hoeken goed zitten voeren we het algoritme uit vanaf de voorste kant. Daarna roepen we de functie opnieuw aan en kijken we opnieuw welke hoeken in de juiste positie zitten. Als er 1 hoek is die in de goede positie zit voeren we het algoritme uit met dit vlak als voorste vlak.

**Bovenste hoeken oplossen**

De laatste stap is het goed draaien van de hoeken zodat ze ook met het gele vlak boven zitten. Dit is een simpel algoritme. Aan de hand van de oriëntatie van het gele vlak kunnen we bepalen hoe vaak dit algoritme toegepast moet worden. Zit het vlak voor dan twee keer, zit het rechts dan 4 keer en zit het al boven dan uiteraard 0 keer. We maken een lijst met het aantal keer dat het algoritme moet worden uitgevoerd per hoek. Dit algoritme begint echter anders dan de andere niet in de hoek die verkeerd zit, maar altijd bij de hoek (positie (0,0,0)) van het voorste vlak. Wanneer het aantal algoritmes is uitgevoerd van 1 hoek dan draaien we het bovenste vlak een kwartslag zodat de volgende hoek in positie komt en voeren we het aantal keer de algoritmes uit voor deze hoek uit.

==== OPTIONEEL ====

**Andere methoden**

Deze methode van oplossen vraagt vaak veel rotaties en veel checks. Daarom duurt dit relatief lang. Er zijn andere methodes die door zogenaamde Speed Cubers competitief worden gebruikt. Het idee is om de code in een latere versie uit te breiden met deze complexere methodes. Dit vraagt minder rotaties en dus minder checks omdat vaak 2 of meer blokjes per keer goed worden gedraaid.

**AI gestuurde methode**

Een andere optie is om een AI te coderen die kan leren om de kubus op te lossen. Ik heb hier nog geen ervaring mee dus wanneer ik hier mee begin zal er eerst een studie moeten plaats vinden over hoe dit precies werkt, en welke methode van AI toepasbaar is op het oplossen van een kubus. Er zijn voor Python wel libraries die het mogelijk maken om AI te coderen. Ik weet echt niet hoe en of dit kan in combinatie met een Arduino microcontroller.

1. Constructie

* **Kubus**

Het eerste waarnaar gekeken moet worden of het logisch is om een originele Rubik’s Cube te gebruiken voor het ontwerp. Wanneer een rotatie net helemaal correct wordt uitgevoerd, kan het zijn dat de kubus vast loopt omdat de zijden niet helemaal loodrecht op elkaar staan. Een oplossing hiervoor zou kunnen zijn om gebruik te maken van een zogeheten speedcube. Deze wordt door professionele cubers vaak gebruikt omdat de rotaties heel soepel zijn én omdat er magneetjes in kubus zitten die ervoor zorgen dat de zijden van de kubus altijd op elkaar aansluiten.

* **Motoren**

Als motoren lijkt de beste optie stappenmotoren te zijn. Deze bevinden zich dan aan elke zijde van de kubus dus in totaal 6 motoren. Het voordeel van aan elke kant een motor is niet alleen dat er zonder ombouw meerdere vlakken meteen gedraaid kunnen worden, maar ook dat de vlakken die op dat moment niet draaien in positie worden gehouden door dor motortjes. Ook kunnen stappenmotoren makkelijk aangestuurd worden door een Arduino.

* **Camera**

Camera moet contrast kunnen waarnemen om de kleuren te onderscheiden. Omdat blokjes meerdere oriëntaties kunnen hebben moet je van ten minste drie kanten kunnen waarnemen welke kleur ze zijn. In een ideale opstelling hebben we 6 camera’s. Voor elk vlak 1. De camera’s worden ook aangesloten op de Arduino om zo de kleurverschillen tussen de verschillende blokjes waar te nemen.

* **Frame**

Moet uiteraard stevig zijn om niet de kubus uit balans of uit positie te brengen. Er moet voldoende ruimte zijn om de motoren te huizen. Daardoor moet ook niet het zicht van de camera’s belemmerd worden. (Zie schets in bijlage). Ook moet de Arduino en de printplaat met de aansturing (en evt. adapters) voor de motoren en de camera’s  
Om het frame genoeg stabiliteit te geven lijkt aluminium een goede en bewerkbare optie.  
Er is al een frame gebouwd door eerdere leerlingen. We gaan kijken of we die kunnen gebruiken.

==== OPTIONEEL ====

* **Belichting**

Om de zichtbaarheid van de camera(‘s) te ondersteunen worden led-strips in het frame geïnstalleerd. Deze zijn zo zuiver wit mogelijk

1. Arduino

De Arduino heeft een eigen programmeertaal wat lijkt op C++. Ik heb geen ervaring met beide dus hiervoor zal tijd ingepland worden om hier dingen over te leren. Verder zijn er bepaalde types Adruino’s die kunnen werken met iets wat heet Micro Python, maar ook hiervoor geldt dat er tijd moet worden geïnvesteerd in onderzoek naar hoe het werkt. Daarnaast zullen de motoren en de camera’s moeten worden geprogrammeerd zodat deze werken naar behoren. In de Python code was dat niet van toepassing dus ook hiervoor moet tijd worden ingepland.

Wanneer dit is onderzocht moet de python code vertaald worden naar de Arduino.