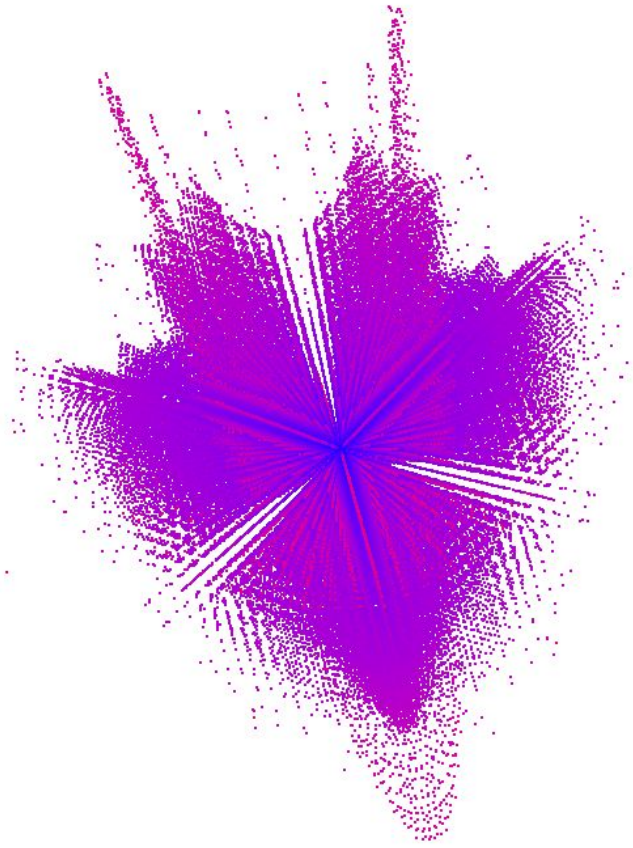
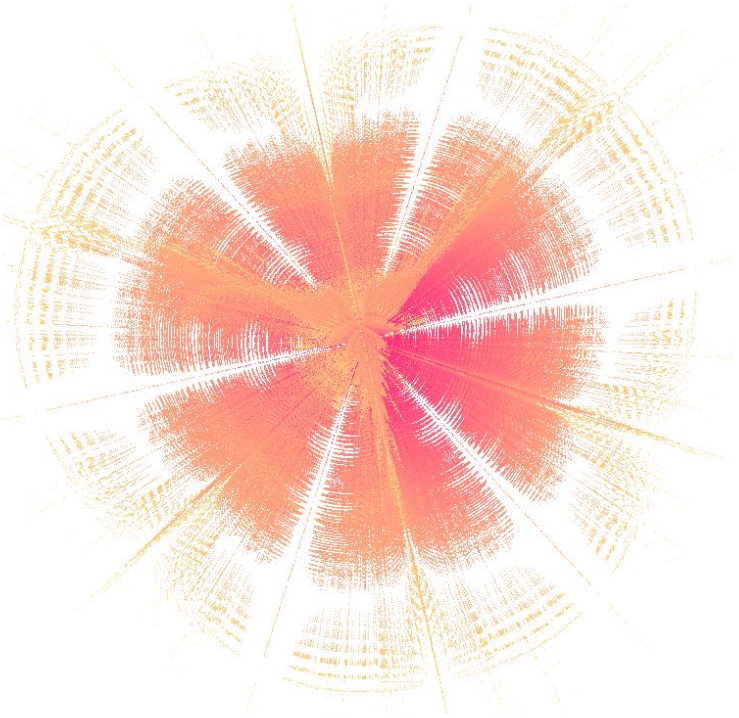
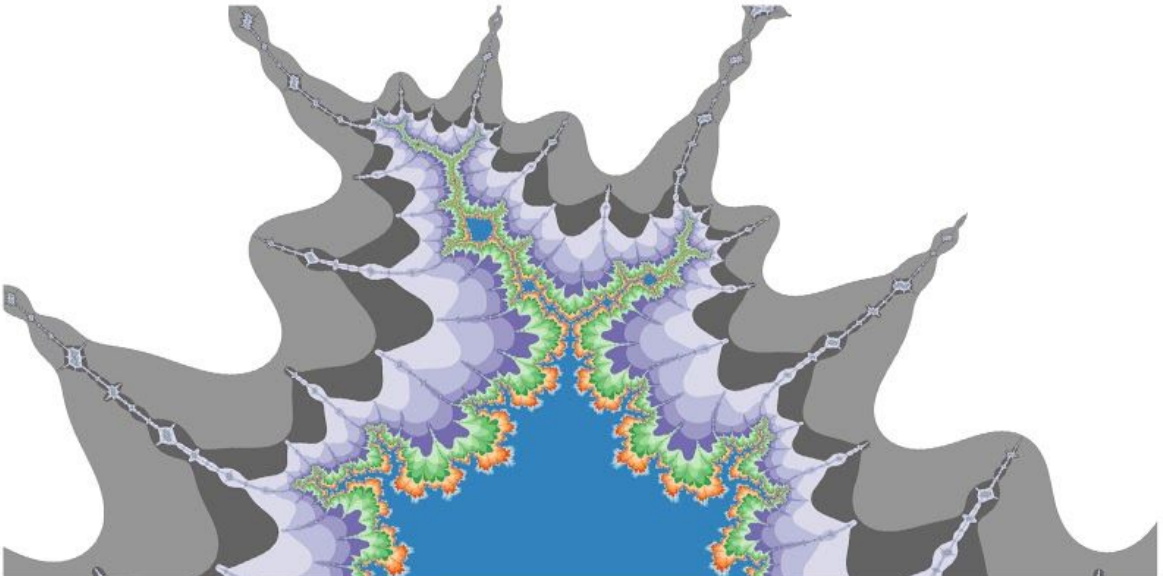
Fractals & Hogere Dimensies

Infinitely Iterated & Intercorrelated Images & Ideas

Geschreven en gecodeerd door Michael Schuchter







Introductie

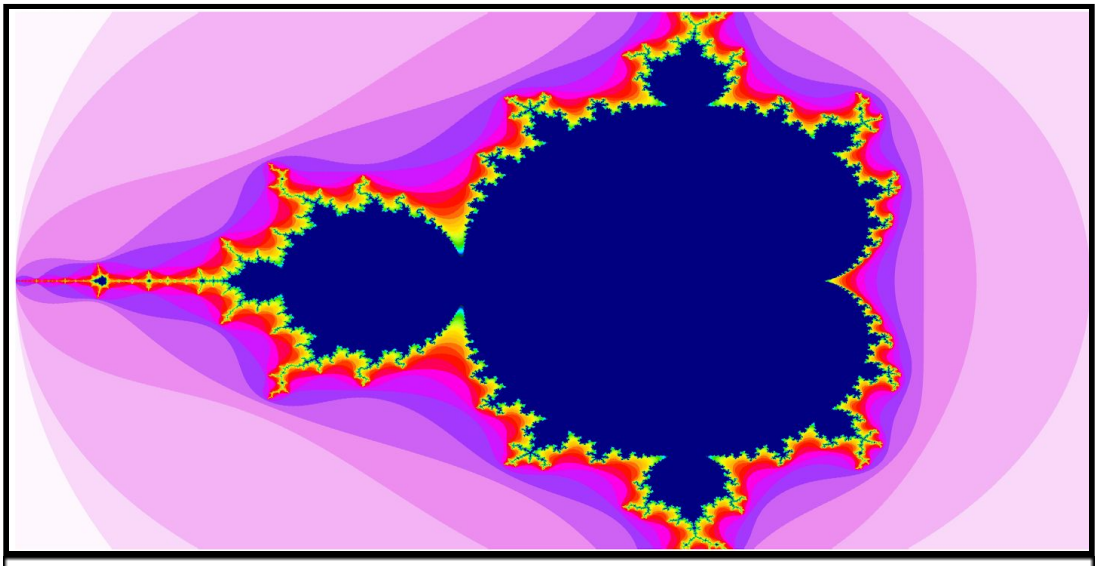
Benoit Mandelbrot was een intelligente wiskundige die in 1975 een nieuwe term introduceerde binnen de wiskunde. Hij stelde het principe van een fractal voor, wat begon met een nogal zware definitie, namelijk:

"Een fractal is bij definitie, een verzameling waar de Hausdorff dimensie de topologische dimensie overschrijdt"

Dat is niet bepaald simpel te interpreteren en dat realiseerde hij zelf ook. Daarom kwam hij later met een wat lossere definitie, namelijk:

"Een fractal is een vorm gemaakt van delen die soortgelijk zijn tot de vorm in een bepaalde manier."

En vanuit daar was het concept van een fractal iets concreets. Hierna, net als fractalen, werd deze nieuwe wiskunde steeds gedetailleerder en uitgebreider naarmate er meer en dieper onderzoek werd gedaan. Met de opkomst van steeds sterkere computers werd het zelfs mogelijk om deze fractalen te laten genereren in steeds hogere resolutie en kwaliteit. Tijd gaat natuurlijk vooruit, en dat brengt ons naar het heden. Naar dit project, en mijn profielwerkstuk.
Dit project zal gaan over het bestuderen van hogere dimensionale fractalen, en wat we kunnen met dit soort wiskundige objecten. Het idee om dit te gaan doen komt vanuit mijn wiskundige interesse, waardering voor mooie figuren, en de oneindige diepte van fractalen.
Onder begeleiding van Mevrouw I. Loeb, het internet, en mijn vriend wie programmeur is, zal ik dit onderzoek gaan houden. Deze begeleiding gecombineerd met mijn interesse voor het onderwerp maakt mij erg gemotiveerd voor het project, en daarom gaat het onderzoek nu beginnen.



De Mandelbrot Set, gegenereerd vanuit eigen codering.
Iteraties per punt: 256
Resolutie: 3000x2000

Inhoudsopgave

Introductie **1**

Inhoudsopgave **2**

Inleiding van het onderzoek **5**

Onderwerp 5

Onderzoeksvragen & Hypothese 6

Aanpak 7

Vooronderzoek **7**

Python 8

De grootste limitatie van Python 8

Toekomstbestendigheid 10

Python Rendering 11

Pillow 11

Matplotlib 11

OpenGL 11

Fractals 12

Dimensie 12

Fractale Dimensie 12

Kustlijnen en het meten daarvan 13

Generalisatie van het concept 14

De fractale dimensie van een Sierpinski Driehoek 15

Fractale eigenschappen 16

Generatieve methoden 17

Iteratieve functie systemen 17

Vreemde aantrekkers 18

Ontsnappingstijd fractals 21

Conclusie van het vooronderzoek **23**

Wat is een fractal, en welke eigenschappen heeft dit wiskundige object? 23

Hoe wordt een fractal gegenereerd, en hoe kan dit generatie process worden beïnvloed? 23

Hoe kan het gebruik van hogere dimensionale datasets worden geoptimaliseerd? 23

Het hoofdonderzoek **24**
Inleiding 24
Generatie van de multidimensionale M.set 24
De absens van driedimensionale complexe getallen 25
Associatieve delingalgebras 25
Het restant van de bewering 26
Hoe de multidimensionale generatie werkt 26
Quaternions 26
De generatie door middel van Quaternions 27
Coderen van de generatie 28
De eisen van het programma 28
Het generatieve programma 29

Projecties van de M.set 33
Projecties door middel van lineaire algebra 34
Lineaire algebra in Python 35
Projecties door middel van geometrie 38
Geometrische projectie in Python 41
Conclusie over projecties 44
Visualisatie van de set 44
Het gebruik van OpenGL en de globale werking 44
Programma & Hoofdstuk Structuur 45
Het hoofdprogramma 46
Dataset Generator 47
De opzet van een glfw 'window' 48
Shaders 50
De stappen van de OpenGL render procedure 50
Vertex Shader 52
Fragment Shader 53
Shader Programma 54
Opzet van OpenGL data 55
De 'Render' functie 57
De 'Main' functie 60
Conclusie 61
Hogere resolutie 61

Optimalisatie van het programma 63
OpenGL buffers 63
Speciale functies van numpy 63
Management van data 63

Conclusie **64**
Deelvragen 64
Wat is een fractal, en welke eigenschappen heeft dit wiskundige object? 64
Hoe wordt een fractal gegenereerd, en hoe kan dit process worden beïnvloed? 64
Hoe kan het gebruik van hogere dimensionale datasets worden geoptimaliseerd? 65
Hoofdvraag 65
Hoe is een multidimensionale fractal te visualiseren in een lagere dimensie? 65
Discussie **66**
Dankwoord **66**
Een laatste woord **67**
Abstract **67**
Literatuurlijst **68**
Experts: 68
Literatuur 68
Plan van Aanpak **70**
Logboek **71**

Inleiding van het onderzoek

Onderwerp

Dit onderzoek zal gaan over multidimensionale fractalen. Dat houdt dus in dat het onderzoek gaat over een object dat een bepaalde fractale eigenschap bezit, in een, of meer dimensies. Mijn voorbereiding voor dit onderzoek heeft me al vrij veel onderzoek laten doen naar de tweedimensionale Mandelbrot Set. Een van de bekendste fractalen van de hedendaagse fractale geometrie. Deze verzameling is erg rijk aan detail, en relatief makkelijk te generen. Daarom leek het mij een goed beginpunt voor mijn uiteindelijke doel.
Het doel wordt om een vierdimensionale Mandelbrot Set te maken, en daar onderzoek naar te doen. Dit onderzoek zal voornamelijk van wiskundige aard zijn, maar een groot deel van het onderzoek zal ook computer gerelateerd zijn. Dit komt voort vanuit het feit dat het genereren van fractalen met de hand vrijwel onmogelijk is, papieren Dragon Curves uitgezonderd.

Draken Curve geproduceerd door papieren stroken op een bepaalde wijze te vouwen

Nu kan de vraag opkomen: "Maar waarom een vier-dimensionale fractal?". Deze keuze is genomen omdat de tweedimensionale Mandelbrot Set al is onderzocht tot een zekere diepte, net als vele andere "bekende" fractalen. Natuurlijk is de logische stap dan om voor een drie-dimensionaal alternatief te gaan, maar dit is waar wiskundige limitaties opkomen. Er bestaat namelijk geen oneven-dimensionale ruimte opgebouwd uit complexe getallen. Een wiskundige uitleg hierover is te lezen in *"De absens van driedimensionale complexe getallen"*. Dus leek het mij een goed idee om voor een extravagant alternatief te gaan, namelijk een vierdimensionale set, die *wel* mogelijk is.

Uiteindelijk zal het eindproduct van dit onderzoek een aantal computerprogramma's zijn, die zelf zullen worden geschreven. Samen met dit verslag; waarin alle ondervindingen en resultaten van het onderzoek verwerkt in zijn. Daarmee zal dit onderzoek afgerond zijn.

Onderzoeksvragen & Hypothese

De hoofdvraag van dit onderzoek zal zich bezighouden met de visualisatie van de multidimensionale fractal naar een lagere, en daarmee zichtbare, dimensie. De vraag die hieruit ontstaat is als volgt:
Hoe is een multidimensionale fractal te visualiseren in een lagere dimensie?

Om te weten wat er precies wordt bedoeld met een fractal kan ook geen kwaad. Dus de vraag:
*Wat is een fractal, en welke eigenschappen heeft dit wiskundige object***?**

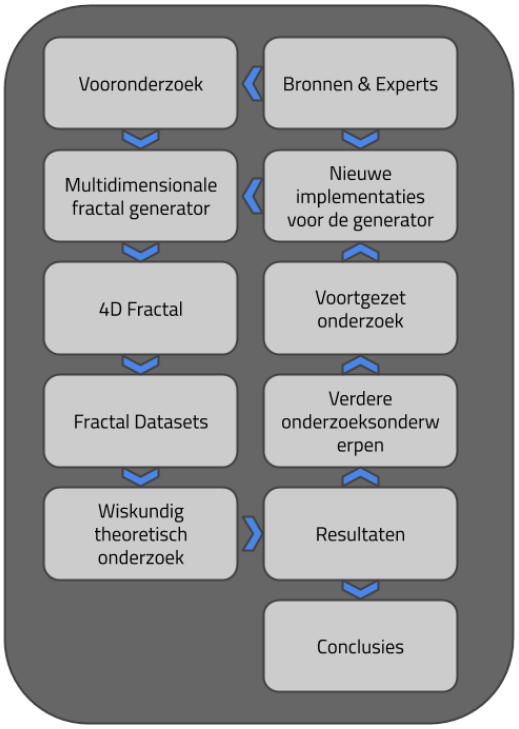
Fractals komen niet uit het niets, dus moet er ook worden onderzocht hoe een fractal gegenereerd kan worden, en wat invloed heeft op de generatie. Daarom:
Hoe wordt een fractal gegenereerd, en hoe kan dit generatie process worden beïnvloed?

Tot slot is er nog de kwestie van hoe er efficiënt met multidimensionale datasets kan worden gewerkt. Mijn thuiscomputer is zeker wel een krachtige rekenmachine, maar zelfs met de tweedimensionale Mandelbrot Set duurde het even voordat de plaatjes klaar waren. Dus:
Hoe kan het gebruik van hogere dimensionale datasets worden geoptimaliseerd?

De hypothese voor de hoofdvraag is gebaseerd op mijn kennis verkregen vanuit de wiskunde D lessen over lineaire algebra. Daar werd er gewerkt met matrix transformaties en projecties. Deze projecties kunnen worden gebruikt om van een driedimensionaal object een tweedimensionale projectie te maken, dus:
"Het is mogelijk om de multidimensionale objecten naar een lagere zichtbare dimensie te brengen door middel van lineaire algebra. Dit wordt gedaan door middel van een matrix transformatie. De eigenschappen van de fractal die bewaard blijven in de lagere dimensie, is afhankelijk van de soort matrix die wordt gebruikt."

Aanpak

Het werkstuk zal worden opgebouwd vanuit verschillende onderdelen. Het begint bij het vooronderzoek, en bestaat uit het onderzoeken van de deelvragen.
Als de deelvragen eenmaal zijn beantwoord, zal de eerste versie van de fractal generator worden opgebouwd. Hiermee kunnen fractal datasets worden gegenereerd, en potentieel beelden hiervan. Het onderzoek zal zich vervolgens voortzetten door deze datasets te onderzoeken. Het resultaat van dit onderzoek kan dan weer leiden tot nieuwe eisen van het programma. Die dan zullen worden geïmplementeerd, en gedocumenteerd.
Dit process van genereren, onderzoeken, verwerken, en implementeren zal worden herhaald totdat alle deelvragen voldoende zijn beantwoord. Vanuit de antwoorden van de deelvragen kan de hoofdvraag worden beantwoordt. Het process van onderzoek is in zekere mate similar tot het genereren van een fractal: Kies een punt, itereer op het punt, en verkrijg een resultaat eruit, en herhaal dit process tot het resultaat voldoende is.



Vooronderzoek

Het vooronderzoek bestaat uit het onderzoeken van Python codering, en specifieker: Hoe het mogelijk is om met Python fractals te genereren en visualiseren. Visualiseren is hier in een andere context dan in de wiskundige context, omdat het visualiseren binnen Python discreet gaat over hoe een computer vanuit data een beeld naar de gebruiker kan overbrengen. Daarentegen is het binnen wiskundige context meer gericht op het behoud van fractale eigenschappen wanneer hogere dimensies worden gereduceerd naar lagere dimensies. Om deze reden zal het visualiseren in computer context worden vervangen door “renderen”.
Ook zal het vooronderzoek bestaan uit onderzoek naar fractals, en de generatie daarvan. Omdat er, ondanks dat dit nog niet bewezen is, vrijwel oneindig veel verschillende manieren zijn om fractalen te generen. En het daarom belangrijk is om een goed beeld te hebben over hoe de generatie zal verlopen van fractals.
Alles overziend zal het vooronderzoek dus bestaan uit een onderzoek over Python codering, hoe Python gebruikt kan worden voor het renderen van fractals, hoe fractals gegenereerd worden en wat hier invloed op heeft, en andere relevante informatie tot deze onderwerpen. Met deze informatie zullen de deelvragen van het onderzoek worden beantwoordt.

Let’s get into it

Python

De programmeertaal Python is uitgebracht in 1991 door Guido van Rossum, en enorm flexibel in wat het kan betekenen voor de programmeur. Codering in het algemeen is een brug tussen een gebruiker en een computer, en manipuleert de computer tot de taak die de gebruiker opgeeft. Dit kan op verschillende niveaus gedaan worden binnen de computer, van het impliciet coderen van de bits in de computer, tot een "High-level programming language" zoals Python. Dat impliceert dat Python codering is gemaakt om goed leesbaar te zijn voor mensen, en niet perse voor computers.
Dit heeft een aantal voor- en nadelen. Het voornaamste voordeel is dat Python enorm gebruiksvriendelijk is, en zeer weinig achtergrondkennis vereist voor gebruik. Hierdoor is de taal erg toegankelijk voor beginners. Ondanks de toegankelijkheid is Python absoluut niet gelimiteerd in wat het kan doen. Python biedt alsnog de diepte die nodig is voor complexere systemen die voorkomen binnen computerwetenschap gerelateerde onderwerpen.

De grootste limitatie van Python

Een nadeel dat zich voordoet in Python is het feit dat de taal meestal trager is dan andere talen, zoals Java, C++, Fortran, en dergelijke. Dit is gebaseerd op de presentatie van een van de Python ontwikkelaars: "Losing your loops" uit 2015. In Python is deze vertraging zeker te merken wanneer de gebruiker te maken heeft met grote datasets, bijvoorbeeld dus fractalen. Dit probleem wordt veroorzaakt door iets genaamd "Type-checking". Dit is het process dat de Python intepeter afloopt naarmate het programma loopt. In dit process worden de types van data gecheckt, voorbeelden hiervan zijn integers, floats, booleans, enzovoort. Omdat deze check uiteraard tijd en computerkracht kost kan een zeer grote verzameling aan data een enorme tijdsbuffer toevoegen aan het programma. Uiteraard zijn er manieren om hier om heen te komen, een van de meest populaire technieken voor het verwerken van veel data binnen Python is via Numpy.

Numpy maakt gebruikt van "arrays", deze arrays zijn stukken in het geheugen van de computer die direct kunnen worden gemanipuleerd, volgens de Numpy documentatie. Dit geeft de gebruiker de mogelijkheid om een grote verzameling aan getallen, ofwel data, te beïnvloeden per array, in plaats van per getal. Hierdoor zorgt numpy voor extreme versnellingen binnen data-intensieve programma's.

```
#De TimeIT module is een module die exact bijhoud hoeveel tijd er
#voor een stuk code nodig is geweest om uit te voeren.
#Dit gebeurt door de code in een string te zetten, en dan uit te voeren
#in een speciale Timeit command.
import timeit
import numpy as np

#Python code string
python_code = """
list = []
```

```
for n in range(1000000):
    list.append(n)
"""
#Numpy import code string
#Dit is nodig om Timeit te vertellen dat numpy geïmporteerd moet worden
numpy_code_import = """
import numpy as np
"""

#Numpy code string
numpy_code = """
list = np.arange(1000000)
"""

#Timeit Test van Python vs Numpy
#Python timeit
print("Python code executie tijd")
python_time = timeit.timeit(stmt = python_code, number = 10)
print(f"{python_time}s")

#Numpy timeit
print("Numpy code executie tijd")
numpy_time = timeit.timeit(setup = numpy_code_import, stmt= numpy_code,
number = 10)
print(f"{numpy_time}s")

#Vergelijking van tijden
ratio = python_time / numpy_time
print(f"Numpy is met deze setup {ratio}x zo snel als python!")
```

Het resultaat van deze benchmark is:

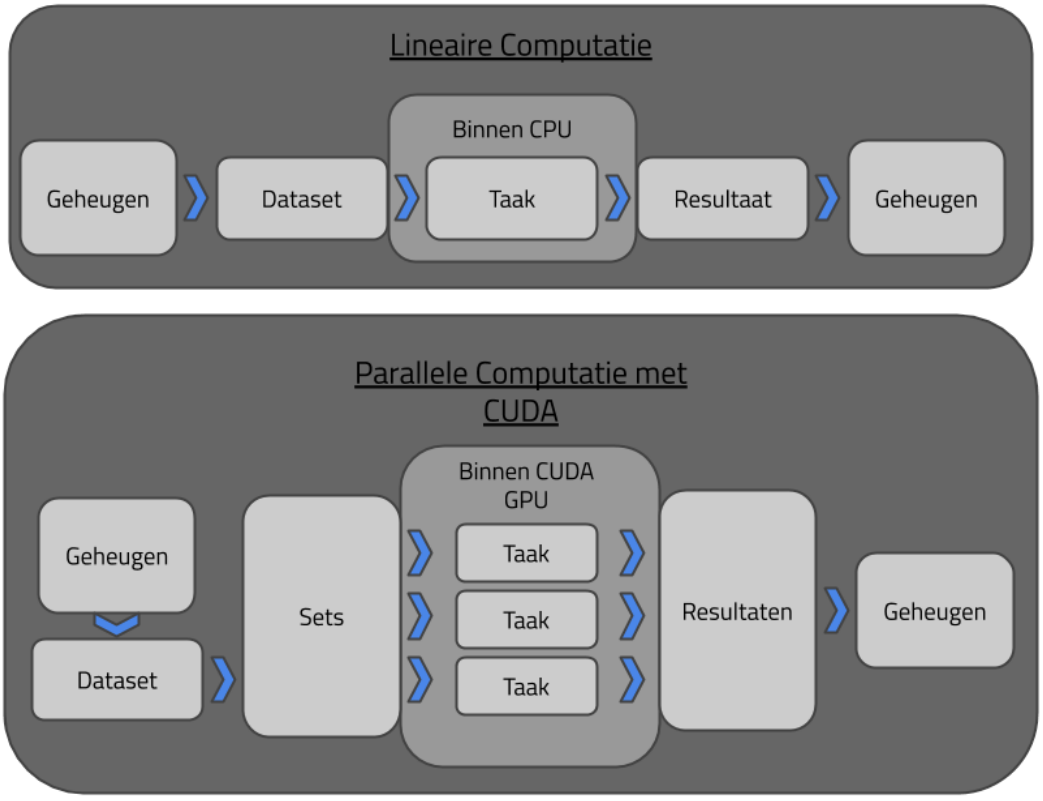
```
Python code executie tijd
1.5303953s
Numpy code executie tijd
0.03279670000000001s
Numpy is met deze setup 46.66308805459084x zo snel als python!
```

Numpy is dus een enorme toename in snelheid, en biedt ook nog eens veel flexibiliteit met wat er gedaan kan worden. Zoals te zien is in de codering, is er voor de Numpy Codering String slechts een lijn aan code nodig. Terwijl in normaal Python een for-loop moet worden gebruikt.

Numpy kan dus zeer veel data in een korte tijd verwerken, maar om absolute zekerheid te garanderen dat het nog snelheid geen knelpunt wordt, is er ook gekeken naar CUDA. Een module voor Python die bliksemsnel is door de hardware van de computer anders te gebruiken. Een diepere uitleg hierover volgt op de volgende pagina.

Toekomstbestendigheid

CUDA is een module die Python codering kan omzetten naar codering die “CUDA-GPU’s” kunnen begrijpen, volgens Nvidia, de uitbrenger van CUDA. Het resultaat van deze GPU’s is dat data-intensieve taken in parallel kunnen worden uitgevoerd. In de de illustratie hieronder wordt er een beeld gegeven hiervan.



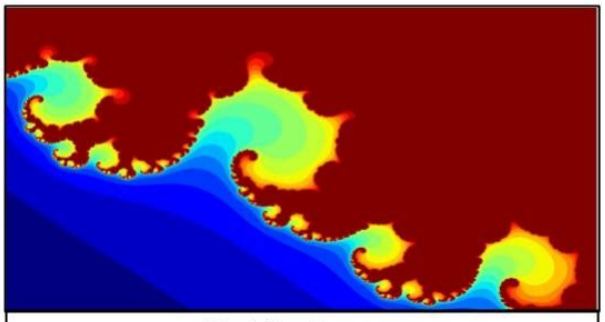
Tijdens lineaire computatie wordt er een taak van het programma continue naar de CPU (Central Processing Unit) gestuurd, waar de taak vervolgens wordt uitgevoerd. Het verkregen resultaat van deze computatie wordt terug gezet in het geheugen, en de cyclus herhaalt. Dit gebeurt totdat het programma ergens vastloopt, of geen resterende taken meer heeft.
In tegenstelling tot lineaire computatie werkt CUDA met de GPU (Graphics Processing Unit). In een gemiddelde computer heeft de GPU meestal meer kernen dan een CPU. Deze kernen zijn meestal minder snel dan een CPU. Maar dat is niet relevant wanneer het om simpele berekeningen gaat, zoals het berekenen van de locatie van een pixel op het scherm. Dus wordt van een grote verzameling aan data kleinere data toegekend aan de kernen in de GPU. Deze kernen voeren een relatief simpele taak uit zoals een geometrische berekening, en leveren vervolgens het resultaat terug aan het geheugen. Dit maakt het computeren van beelden, fractalen, video’s, en meer een stuk efficiënter binnen een GPU dan in een CPU.
Indien het nodig is om de manipulaties van de multidimensionale fractalen te versnellen is CUDA dus de oplossing daarvoor.

Python Rendering

Het genereren van een fractal is relatief simpel indien de juiste regels worden gecodeerd. Daarentegen is het visualiseren van een fractal een stuk moeilijker. Dit komt door de oneindige details die fractalen bezitten, en helaas is er nog geen toegang tot oneindige computatie kracht of geheugen. Dus moet de visueel van een fractal worden gecomprimeerd tot iets dat kan worden weergegeven op een scherm.

Pillow

Python heeft een aantal opties voor het genereren van plaatjes, en renders, volgens de documentatie. Een van deze opties is Pillow. Dit is een module die ervoor zorgt dat Python afbeeldingen kan bewerken, en genereren. De module is gedeeltelijk gecodeerd in C, een taal die significant sneller is dan Python, daarom is de module relatief snel. Helaas ondersteunt Pillow geen driedimensionale rendering, waardoor het afvalt voor dit onderzoek. Daarentegen is het wel erg bruikbaar gebleken voor het genereren van twee dimensionale Mandelbrot Sets, en zooms van de Set, zoals te zien in de *galerij* in de bijlage, en hiernaast.



Mandelbrot Set zoom.
Gegenereerd met de Pillow module van Python

Matplotlib

Verder is het mogelijk om met Matplotlib data visualisaties te maken, ook volgens de documentatie. Hierbij kan men denken aan plots, grafieken, en dergelijke. Matplotlib is relatief snel in twee dimensies, en was daarom erg geschikt voor het vooronderzoek. Maar helaas is de snelheid van de module te laag om real-time renders te maken. Dus valt Matplotlib ook af voor het genereren van de multidimensionale set.
Matplotlib en Pillow zijn beide dus niet geschikt voor het verdere onderzoek. En omdat deze twee modules vrijwel de populairste modules zijn om data mee te visualiseren moest er verder en dieper worden gekeken. Bijvoorbeeld naar Vispy, dit is ook een module om data mee te visualiseren. Ondanks dat deze module erg uitgebreid en krachtig is, was er reden om te zoeken naar iets met nog meer flexibiliteit. De reden daarvoor is dat het programma dat de multidimensionale fractalen zal genereren ook nog eens direct het beeld van de fractal moet kunnen manipuleren. Het is dus belangrijk om een programma te maken die deze mogelijkheid geeft.

OpenGL

Uit intensief onderzoek, en advies van een vriend die programmeur van beroep is, blijkt dat OpenGL de juiste vrijheid geeft die nodig is. Om dan iets specifieker te zijn, OpenGL geïntegreerd met Python. OpenGL staat voor Open Graphics Library. Het is een systeem om 2D en 3D renders te maken, en het is enorm vrij in wat er mee gedaan kan worden. Het enige nadeel is dat de logica, syntax, en taal van OpenGL geleerd moet worden, hiervoor wordt er gebruikt gemaakt van een online gids geschreven door Nicolas P. Rougier. In het hoofdstuk "*Visualisatie van de set*" zal OpenGL uitgebreid worden behandeld.

Fractals

Voordat er kan worden gekeken naar de generatie van een fractal moet eerst wiskundig worden gedefinieerd wat een fractal is. En voordat dat kan worden gedaan moeten er eerst wat andere begrippen worden gedefinieerd.

Dimensie

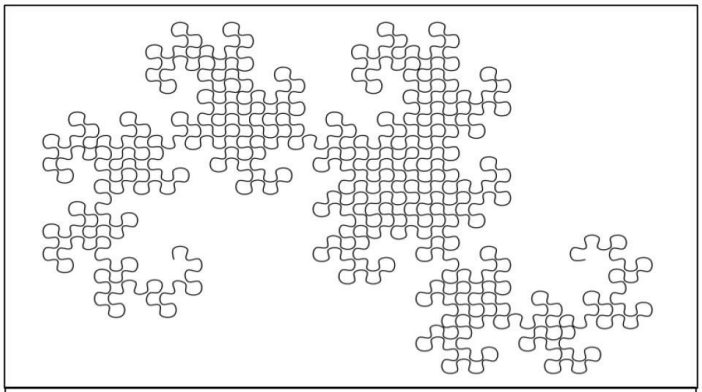
Het universum, voor zover wij het kennen, is drie-dimensionaal. Het scherm, of papier waarop dit verslag wordt gelezen is tweedimensionaal, en de letters bestaan uit lijnen die één dimensionaal zijn. Maar als wiskundige is het belangrijk dat concepten zo abstract mogelijk worden beredeneerd. Dus hoe zou een wiskundige naar het concept van "dimensie" kijken?
Er zou kunnen worden gekeken naar het aantal getallen die nodig zijn om een punt in de gewilde ruimte te beschrijven, ofwel coördinaten. Dit klinkt relatief simpel, maar helaas is er aardig wat topologie nodig om deze definitie formeel te beschrijven. Om dit onderzoek niet te zwaar op topologie te richten wordt daarom elke dimensie die wordt beschreven, gegeneraliseerd tot een metrische ruimte. Volgens Juan Pablo Xandri moet een metrische ruimte aan vier eisen voldoen, namelijk:

- De afstand tussen twee punten is altijd positief.
- De afstand van punt A naar punt B, is hetzelfde als van punt B naar punt A.
- De afstand van een punt naar zichzelf is nul.
- De directe afstand van punt A tot punt B, is gelijk of groter dan de afstand van punt A tot punt C tot punt B

Zolang de fractals zich in ruimtes bevinden die deze "regels" volgen, kan de dimensie worden gegeneraliseerd naar *n*-dimensionaal, waar n een integer getal is.
Met een gegeneraliseerde definitie voor dimensie kan er worden gekeken naar de definitie van de zogeheten "fractale dimensie".

Fractale Dimensie

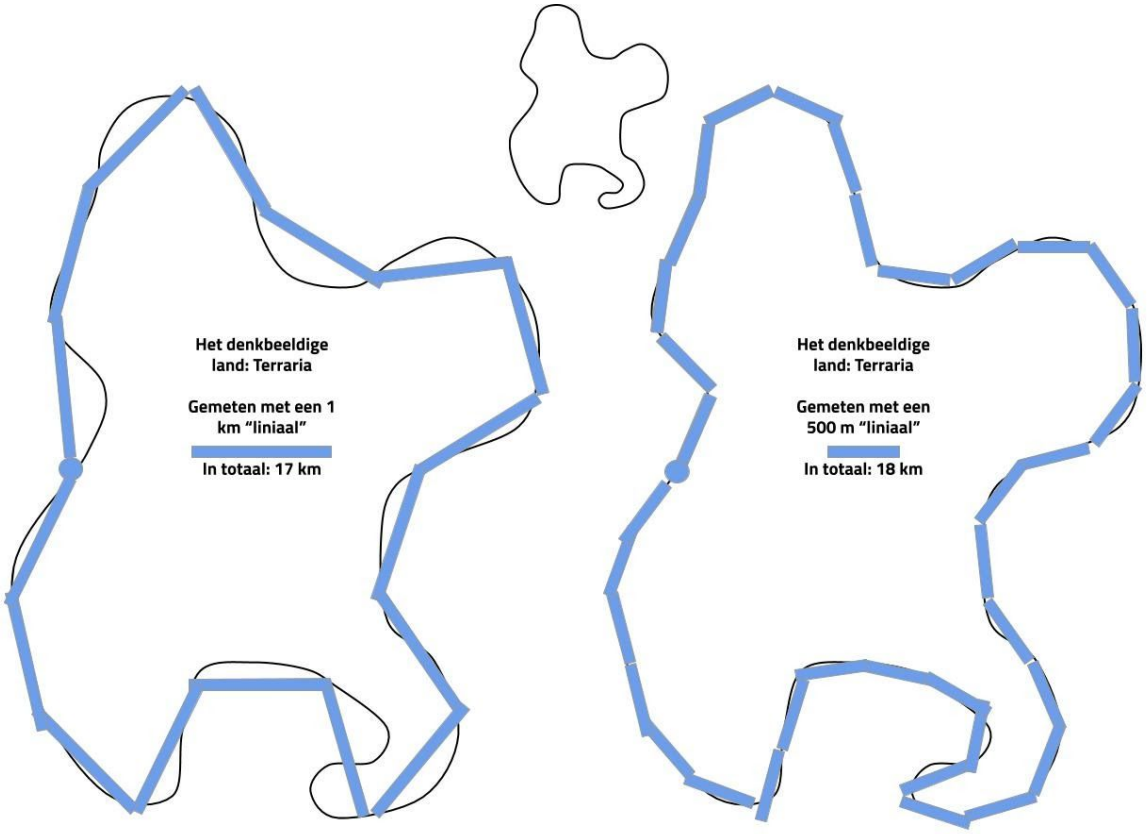
Het concept van een fractale dimensie is nogal abstract. Het houdt zich bezig met hoe gedetailleerd een fractal is naarmate erop wordt ingezoomd, en hangt hier vervolgens een getal aan. En dit getal is groter of kleiner dan de dimensie waar de fractal zich in bevindt, of zelfs gelijk aan. Het heeft ook een betekenis in de zin van hoeveel ruimte een fractal vult in de dimensie waarin het zich bevindt. Zo heeft een Draken Curve, volgens Wikipedia, een fractale dimensie van 1.5236… Hoe dit getal wordt verkregen wordt besproken in "*Generalisatie van het concept*". Maar om een intuïtief gevoel voor dit concept te verkrijgen kan er worden gekeken naar een praktisch voorbeeld. Kustlijnen zijn namelijk ook fractal in een zekere zin.



Een Draken Curve
Bron: Wikipedia

Kustlijnen en het meten daarvan

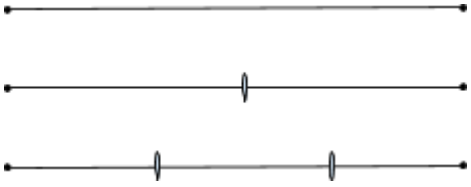
Stel dat iemand een kustlijn zou willen meten, hoe zou deze persoon dat doen? Hij of zij zou kunnen beginnen op een punt op het strand van een land, volgens Steve Mould. En vervolgens met een liniaal van een meter steeds de kust aflopen, en meten, totdat diegene uiteindelijk bij het originele punt aankomt. Vervolgens heeft deze persoon een getal, namelijk het aantal keer dat de liniaal "paste" in de kustlijn. Voor het gemak van het voorbeeld is de gemeten lengte van de kustlijn van dit denkbeeldige land rond de 2500 linialen, ofwel 2500 meter. Dat is top, maar wat als iemand anders nu ditzelfde process herhaalt, maar dan met een liniaal van een halve meter? Het zou logisch zijn dat de kustlijn dezelfde lengte blijft, maar omdat de mate waarmee detail kan worden gemeten is vergroot, zal de lengte ook groter zijn. Zoals te zien in het voorbeeld hieronder:



Dit komt voornamelijk door stukken van de kustlijn zoals het Zuiden, waar een scherpe curve is. Wanneer deze curve wordt gemeten met een kleinere liniaal, kan dit veel nauwkeuriger worden gedaan. Waardoor de uiteindelijke lengte groter zal worden. Dit concept heet de Kustlijn Paradox. Dus de mate waarmee wordt gemeten, of gekeken, bepaalt de lengte van een kustlijn. Dit kan anders worden verwoord naar de mate van detail is afhankelijk van de mate waarmee het detail wordt gemeten. En dit is waar het principe van een fractale dimensie zich mee bezig houdt.

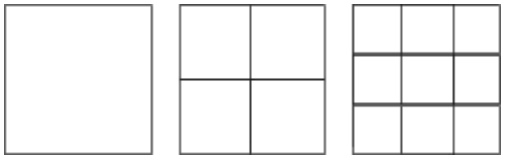
Generalisatie van de Fractale Dimensie

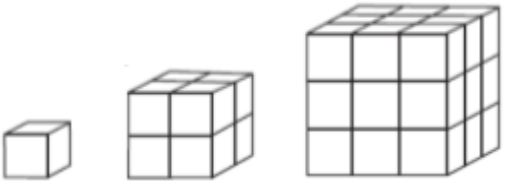
Om hier een wiskundige definitie aan te hangen moet dit concept eerst worden gegeneraliseerd. Dus voordat er naar fractals kan worden gekeken, beginnen we met een simpele eendimensionale lijn. Deze lijn kan worden opgedeeld in twee gelijke stukken, of in *S* gelijke stukken. Dit impliceert dat de lijn met $\frac{1}{S}$ is vermenigvuldigt. En dit kan in een formule worden gezet:



$A = S^{-1}$

Vervolgens kan er met de originele ongedeelde lijn een vierkant worden geconstrueerd. Een vierkant is tweedimensionaal, en kan dus in twee dimensies worden opgedeeld. Bij het delen van het vierkant in vier, negen, of S gelijke kleinere vierkanten volgt het aantal vierkanten een kwadratisch verband. Namelijk: $A = S^{-2}$, waar A het aantal soortgelijke vierkanten is, en S de schaal waarmee de vierkanten worden vermenigvuldigt. Tot slot kan ditzelfde process worden herhaald voor een driedimensionale kubus, die kan worden geconstrueerd uit het originele onverdeelde vierkant. Wanneer de kubus wordt verdeeld, zal het aantal soortgelijke kubussen zich volgens dit patroon uitten: 1, 8, 27, 81, … Dit is een kubisch verband, en kan worden uitgeschreven als: $A = S^{-3}$ Waar A staat voor het aantal soortgelijke kubussen, en S weer voor de schaal waarmee de kubus wordt vermenigvuldigt.





Een oplettende lezer heeft misschien gezien dat er een patroon in de schaal-formules zit. Alle formules volgen namelijk de vorm: $A = S^{-D}$ Waar D voor de dimensie van het object staat, A voor het aantal soortgelijke objecten naarmate de schaal veranderd, en S uiteraard nog steeds voor de schaal.

Deze formule bevat dus een variabel dat staat voor de dimensie waar een object in bestaat, en dat rekening houdt met de schaal waarmee het object wordt bekeken. Dit is dus een zeer toepasselijke formule om de dimensie te achterhalen van een fractal, indien we deze ombouwen:

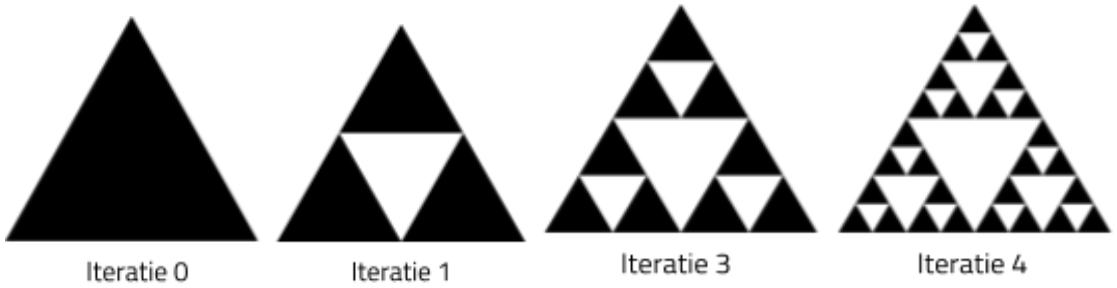
- $A = S^{-D}$
- $Log(A) = Log(S^{-D})$
- $Log(A) = -D * Log(S)$
- $\frac{Log(A)}{Log(S)} = -D$
- $D = -\frac{Log(A)}{Log(S)}$

Met deze formule, een schaal, en een soortgelijk object kan nu de dimensie van het object worden berekend. Dit klinkt allemaal nog erg abstract, dus om deze formule in werking te zien zal de fractale dimensie van een Sierpinski driehoek worden berekend. Deze wiskundige interpretatie is authentiek, en nagekeken op correctheid via Wikipedia.

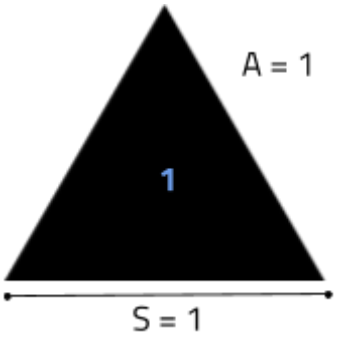
De fractale dimensie van een Sierpinski Driehoek

Een Sierpinski driehoek is een vrij simpele fractal om te genereren:

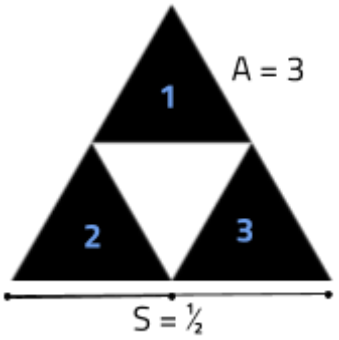
1. De fractal begint met een gelijkzijdige driehoek, ofwel iteratie 0.
2. In deze driehoek, wordt een nieuwe driehoek geconstrueerd. Die de originele driehoek in vier gelijke kleinere driehoeken verdeeld, waarvan drie driehoeken zwart worden gekleurd. Zoals te zien in Iteratie 1.
3. Vervolgens zal elke zwarte driehoek stap 1 opnieuw doorlopen. Waardoor een recursief patroon ontstaat.



Maar hoe kan hieruit een fractale dimensie worden berekend? Relatief simpel, omdat de sierpinski driehoek soortgelijk is aan zichzelf. Het zijn uiteraard alleen maar driehoeken in driehoeken.
Om aan de getallen te komen die nodig zijn voor de formule, ofwel A voor het aantal soortgelijke figuren, en S voor de schaal, moeten er soortgelijke objecten worden geteld en een schaal worden aangelegd.



De zijde van de driehoek in Iteratie 0 wordt als originele schaal genomen, ofwel $S = 1$. En er wordt een soortgelijk object geteld, namelijk de driehoek, dus $A = 1$. Met deze kan de fractale dimensie nog niet worden berekend, omdat er nou eenmaal nog naar een simpele driehoek wordt gekeken. Dus wordt er verder gekeken naar iteratie 1. Hierin is te zien dat de originele schaal van de zijde is verdeeld in twee stukken, ofwel de twee kleinere driehoeken, dus de schaal is: $S = \frac{1}{2}$. Dan kunnen er ook drie soortgelijke objecten worden geteld, ofwel de drie zwarte driehoeken, dus $A = 3$. Met deze getallen kan er wel wat worden berekend:



- $D = -\frac{Log(A)}{Log(S)} \mid A = 3,\ S = \frac{1}{2}$
- $D = -\frac{Log(3)}{Log(\frac{1}{2})} = Log_2(3) \approx 1.5849$

Dus de fractale dimensie van een Sierpinski Driehoek is gelijk aan $Log_2(3)$, ofwel ~1.5849. De fractale dimensie is dus lager dan de dimensie waarin de fractal zich bevindt, namelijk 2D. Indien dit getal wordt bekeken vanuit een ruimtevullend perspectief, betekent het dus dat van de twee dimensies, de Driehoek ongeveer 1.5849 dimensie aan ruimte vult.

In conclusie, het concept van een fractale dimensie kan in twee perspectieven worden bekeken. Namelijk vanuit een ruimtevullend perspectief, en een perspectief van hoe complex de details zijn naarmate de schaal veranderd waarmee wordt gekeken. Om een getal te krijgen voor dit kan er gebruik worden gemaakt van de formule:

$$D = -\frac{Log(A)}{Log(S)}$$

Om de fractale dimensie van een fractal te berekenen wordt er gekeken naar het aantal soortgelijke objecten, op verschillende schalen. Deze getallen worden in de formule gebruikt om de dimensie te berekenen. Deze formule kan echter ook worden gebruikt voor objecten die niet fractal zijn.

Fractale eigenschappen

Deze "fractale dimensie" geeft een goede richtlijn over wat een fractal is, maar beschrijft nog niet de hele definitie. De definitie van een fractal is namelijk niet concreet vast te leggen. Mandelbrot stelde ooit dat een fractal een "gefragmenteerde geometrische vorm, die kan worden gesplitst in delen, die weer lijken op het geheel" is, in zijn boek: "The fractal geometry of Nature". Ofwel wat eerder is geobserveerd bij het beredeneren van de fractale dimensie. Het probleem met deze definitie is dat het te gelimiteerd is. Een andere definitie van Mandelbrot zelf is: "Een fractal is een verzameling aan getallen waarvan de Hausdorff-Besicovitch dimensie groter is dan de topologische dimensie". Ofwel:

$$F \subseteq \mathbb{R}^n \mid n \in \mathbb{Z},\ dim_H(F) > dim(F)$$

Waarin:

- F staat voor de verzameling van getallen van de fractal.
- $dim_H(F)$ Staat voor de Hausdorff Dimensie van de fractal.
- dim(F) Staat voor de topologische dimensie.

En hoewel deze definitie meer wiskundige vrijheid biedt, is het niet een toepasselijke definitie voor ruimtevullende curves. Een tak van fractals die een complete dimensie kunnen vullen door middel van een lager dimensionaal object, zoals een vlak gevuld door puur een lijn.
Dus wat moet de definitie dan zijn? Kenneth John Falconer, een wiskundige in het veld van fractale geometrie, stelt dat fractals niet vast te leggen zijn door middel van een vaste definitie. In plaats daarvan moet het object waar naar gekeken wordt voldoen aan een aantal eigenschappen:

- Soortgelijkheid, met verschillende vormen:
    - Exacte soortgelijkheid, zoals de Sierpinski Driehoek
    - Quasi-soortgelijkheid, ofwel waar het geheel ongeveer wordt gerepliceerd op andere schalen. Dit gebeurt bijvoorbeeld bij de Mandelbrot Set.
    - Statistische soortgelijkheid, waar bepaalde patronen zich herhalen op verschillende schalen.
- Structuren die intrinsiek gedetailleerd zijn, zelfs op extreem kleine schaal.
- Een zekere chaos en onregelmatigheid in het patroon, dat niet direct kan worden beschreven in geometrische uitdrukkingen.
- Generatieve methoden die relatief simpel zijn om uit te voeren.

Deze groep aan eigenschappen geven dus een richtlijn voor wat een fractal zou moeten zijn, zonder bepaalde fractals buiten te sluiten zoals de eerdere definitie dat deed. Daarom zal er in dit onderzoek voornamelijk gebruik worden gemaakt van deze groep aan eigenschappen, in plaats van een concrete definitie.

Generatieve methoden

Fractals kunnen op verschillende manieren worden gegenereerd, sommige methoden maken gebruiken van iteratieve complexe functies, andere methoden kunnen zo simpel zijn als het vouwen van een stukje papier. Om een beter begrip te krijgen over de generatie zullen een aantal generatieve methoden worden behandeld, met nadruk op computergestuurde generatie.

Iteratieve functie systemen

Gebruikt programma: barnsley_fern_generator.py

Fractals die voortkomen uit iteratieve functies volgen, zoals de naam al aangeeft, een functie die wordt ge-itereert over een gewenst aantal iteraties. De functie kan erg complex, letterlijk en figuurlijk, of juist heel simpel zijn. Een goed voorbeeld van een Iteratieve Functie is de "chaos game". Deze chaos game staat beschreven in het boek "Chaostheorie en klimaatverandering" van Carlos Madrid.

Deze chaos game begint door een nulpunt te nemen in het midden van een tweedimensionaal vlak. Dit punt zal worden genoteerd als P. Ofwel:

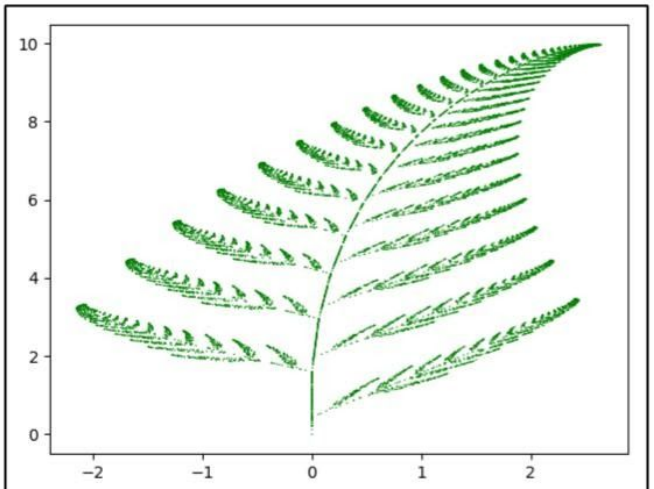
- $P_n = (X_n, Y_n)$

Vanaf hier wordt een puur willekeurig getal van 0 tot 100 gekozen, ofwel een willekeurig percentage. Dit getal zal worden genoteerd als R. Vervolgens zal aan de hand van dit getal een nieuw punt worden geplot volgens de volgende regels:

- Als R = 1:
  - $X_{n+1} = 0$
  - $Y_{n+1} = A*Y_n$
- Als 1 < R < 85:
  - $X_{n+1} = B*X_n + C*Y_n$
  - $Y_{n+1} = C*X_n + B*Y_n + E$
- Als 85 < R < 92:
  - $X_{n+1} = F*X_n + G*Y_n$
  - $Y_{n+1} = H*X_n + I*Y_n + J$
- Als 92 < R $\leq$ 100:
  - $X_{n+1} = K*X_n + L*Y_n$
  - $Y_{n+1} = M*X_n + N*Y_n + O$

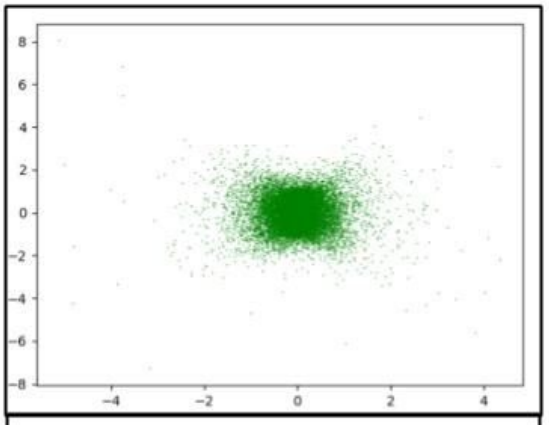
Als dit systeem aan regels een paar honderd of duizend keer wordt ge-itereerd, komt het volgende beeld naar voren:

Dit wordt ook wel het varenblad van Barnsley genoemd, vanwege de vorm. De coëfficiënten voor deze specifieke vorm zijn gebaseerd op de "optimale" vorm, die zijn afgeleid van de natuur.



Een varenblad van Bernsley
Gegenereerd via Python met 15000 iteraties.

Uiteraard is het mogelijk om eigen waardes te kiezen, en verschillende coëfficiënten zullen verschillende resultaten opleveren. Zoals te zien is in de tweede afbeelding, waar alle coëfficiënten willekeurig zijn gegenereerd.
Het resultaat is niet enorm interessant. Het is vrij willekeurig zoals verwacht zou kunnen worden. Hoewel deze afbeelding niet heel interessant is om naar te kijken, impliceert het wel iets over de generatieve eigenschappen van fractals. Namelijk dat de coëfficiënten van de generatieve formule zeer invloedrijk zijn op het uiteindelijke beeld. Dit spreekt ergens voor zichzelf, maar een concreet voorbeeld is nou eenmaal bevestigend.



Een compleet willekeurig gegenereerd varen-bolletje.

In conclusie, een iteratieve functie kan dus gebruikt worden om fractals te genereren. Het is wel belangrijk om hierbij te vermelden dat niet zomaar elke iteratieve functie zal werken. En dat wanneer er coëfficiënten worden gebruikt in de functie, er verschillende resultaten kunnen ontstaan wanneer deze veranderen.

Vreemde aantrekkers

Gebruikt programma: logistic_map_fractal.py
Dynamische systemen zijn een hele aparte tak van wiskunde, maar uiteraard komen er ook fractals voor in deze tak. Voordat er naar de fractal kant worden gekeken moet er eerst wat context worden gegeven over wat een aantrekker is binnen deze wiskundige tak.
Wanneer een dynamisch systeem richting een bepaalde waarde convergeert, wordt dit een aantrekker genoemd. Deze aantrekkers kunnen multidimensionaal zijn, en kunnen een grote variatie aan dingen betekenen. Omdat de meeste aantrekkers multidimensionaal zijn, worden ze meestal uitgedrukt in vectoren. En vanuit deze multidimensionaliteit kan uiteraard ook een multi-dimensionaal beeld worden geproduceerd. Als het geproduceerde beeld fractale eigenschappen heeft, dan wordt de aantrekker *vreemd* genoemd.
Een bekend dynamisch systeem dat fractale eigenschappen heeft is de Logistieke Map. Dit is een functie die bijvoorbeeld gebruikt kan worden om de reproductie van een populatie aan konijnen te voorspellen. De formule die hiervoor wordt gebruikt is in de vorm van:

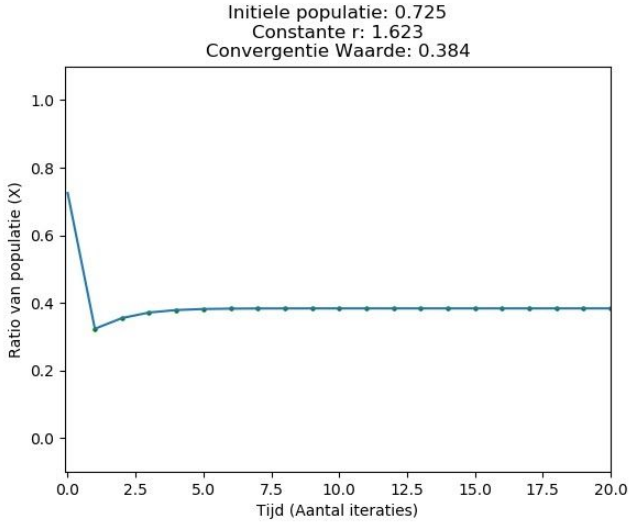
$$X_{n+1} = r * X_n(1 - X_n)$$

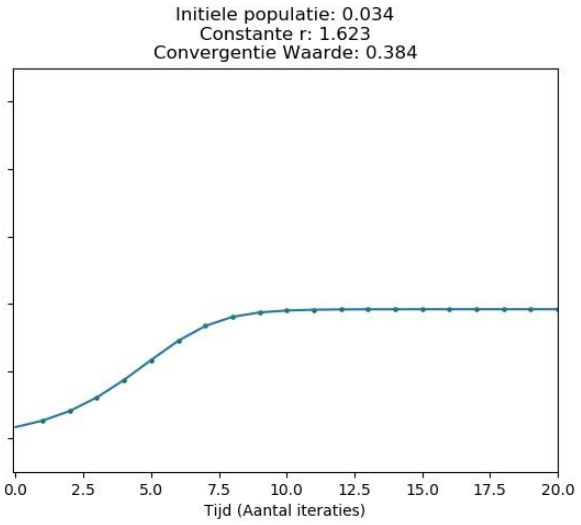
Waarin:

- X_n Een getal is tussen 0 en 1, dat de bestaande populatie representeert als een ratio tot het maximum van de populatie.
- r Is een constante, waarvan de interessante punten liggen tussen de 0 en de 4. Constanten daarboven geven relatief weinig interessants, omdat de formule al snel richting oneindigheid gaat.

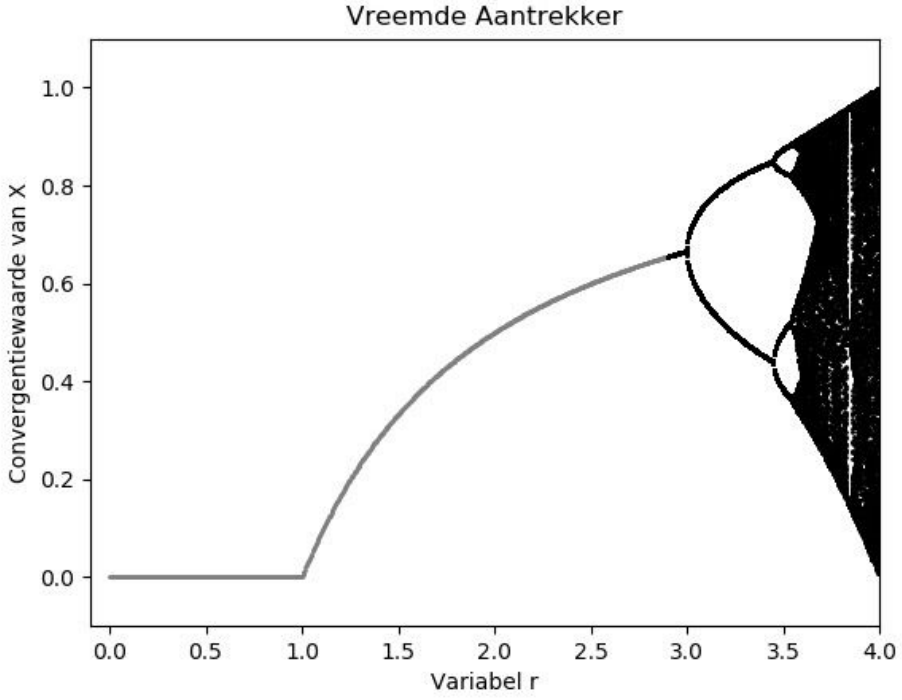
Om bij de fractale kant te komen moeten er een aantal stappen worden gevolgd. Deze stappen zijn zeer goed beschreven door Veritasium in zijn video: "This equation will change how you see the world". En hoewel de titel misschien wat hyperbolisch is, zijn de resultaten alsnog zeer mooi:

1. Om te beginnen wordt X_n tegen het aantal iteraties geplot. Het aantal iteraties van de formule kan als tijd worden gezien. Om deze berekening uit te voeren moet er ook een X_0 worden gekozen. Maar de waarde van X_0 is niet relevant, omdat de uiteindelijke waarde van de grafiek zich naar een evenwicht zal bewegen. Dit evenwicht is de waarde waarnaar de formule convergeert, en deze waarde is belangrijk voor stap 2. Dit process wordt gedemonstreerd in de grafieken hieronder.

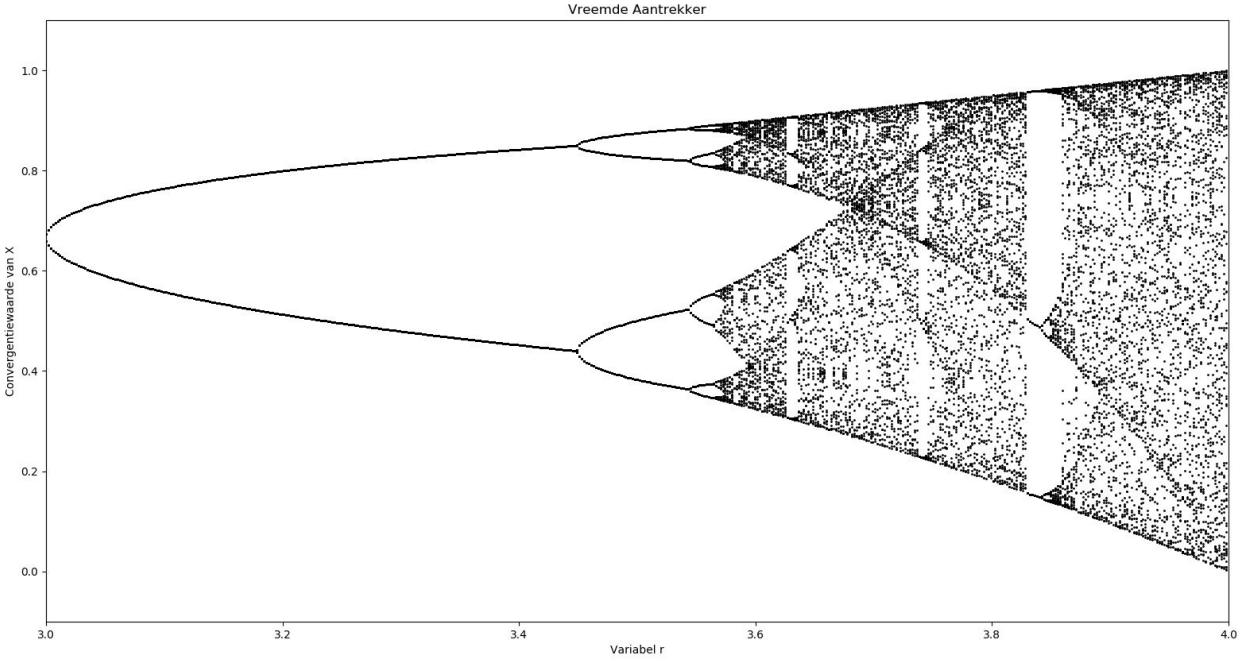




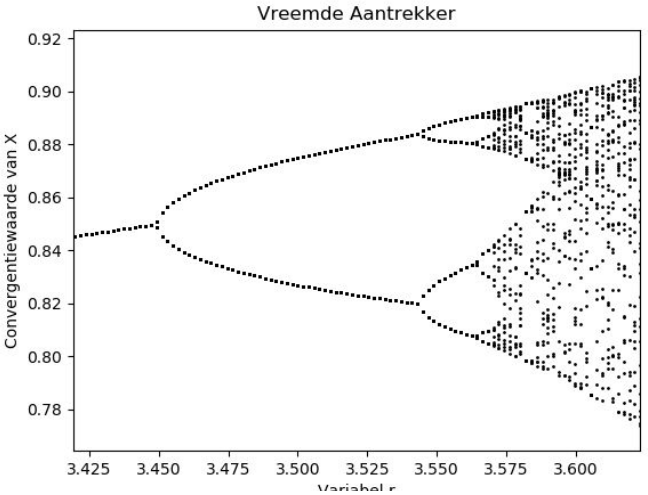
2. Om het fractale deel van deze aantrekker te krijgen moet er in een nieuwe grafiek:
    a. r Als x-coördinaat worden genomen.
    b. De evenwichts populatie van een geïtereerde formule met de gekozen r als y-coördinaat. Het is belangrijk om hierbij te melden dat de initiële populatie op een constante wordt gehouden, in de volgende stappen is dat een waarde van 0.5.
3. Laat de grafiek plotten, en dit is het resultaat:



Indien er wordt ingezoomd op het domein van [3, 4] is dit beeld te zien:



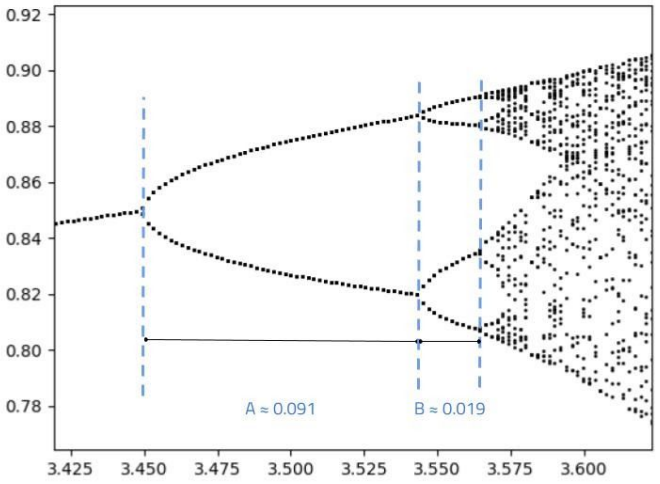
En hoewel deze afbeelding niet de hoogste kwaliteit render ooit is, is er wel op te merken dat het figuur fractale eigenschappen bevat. Namelijk soortgelijke patronen op verschillende schalen, zoals te zien in de afbeelding hiernaast, die op het domein van [3.425, 3.65] is gezoomd. Op de waarde, verkregen via Wikipedia, van $r \approx 3.56995$ begint de functie chaotisch gedrag te vertonen. Waarna het op $r \approx 3.82843$ weer terugkeert naar orde, ondanks dat niet te zien is in de afbeeldingen.



Ook is het opmerkelijk dat deze fractal een constante bevat. Wanneer men de afstand meet van een splitsing tot de volgende, en nog een volgende. En het ratio van deze twee afstanden berekend komt men uit op:

$$\frac{A}{B} = \frac{0.091}{0.019} \approx 4.669 \ = \delta$$

Ofwel de Feigenbaum constante, ontdekt door de heer Mitchell J. Feigenbaum.



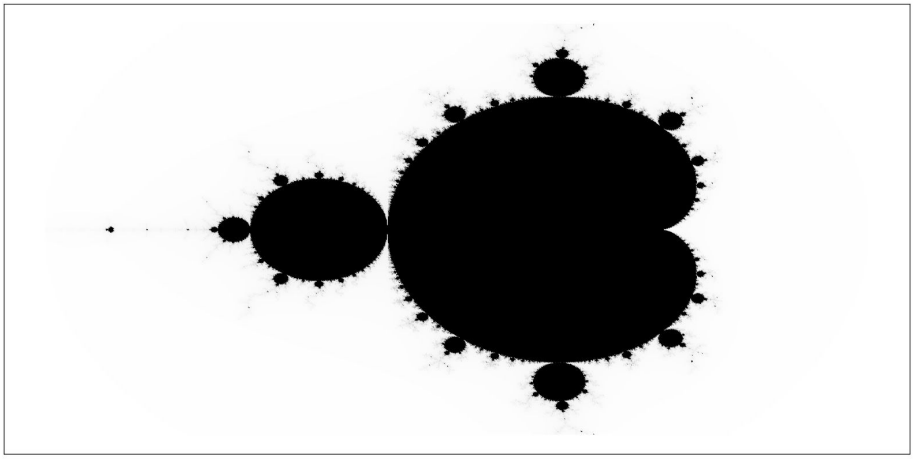
Deze fractal wordt overigens de Feigenbaum Aantrekker genoemd, of het bifurcatiediagram. In conclusie, door middel van Aantrekkers, en het laten variëren van variabelen binnen dynamische systemen kunnen fractals worden gegenereerd.

Ontsnappingstijd fractals

Gebruikt programma: mandelbrot_v3.py

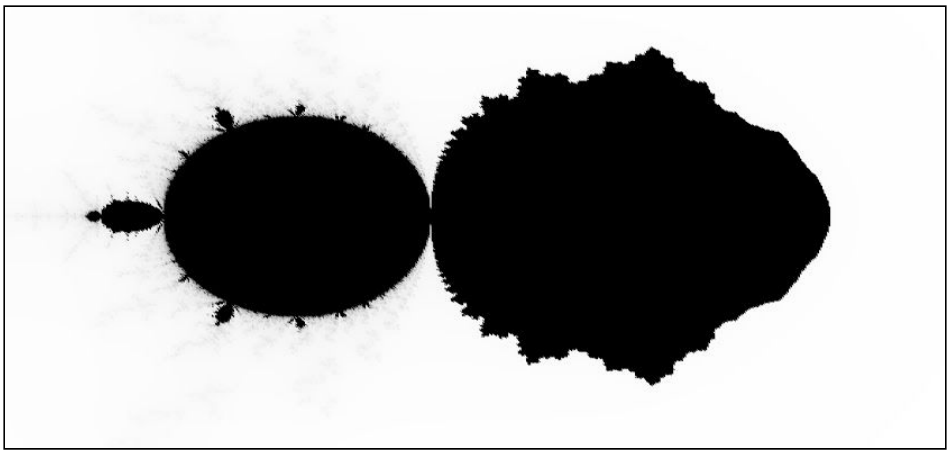
Een ontsnappingstijd fractal is een fractal die ontstaat door op een verzameling van coördinaten een bepaald systeem aan regels, functies, of recursies toe te passen. Dit kan met normale coördinaten gebeuren, of met complexe getallen die uit zichzelf al een coördinaat uitdrukken in het complexe vlak. Een systeem van zulk soort regels kan er uit zien als het volgende. Volgens een zekere Holly Krieger, wiskundige aan MIT :

- Neem een punt uit het complexe vlak, dit is in de vorm van: $c = a + bi$
- Itereer de functie: $F_c(z) = z^2 + c$, met c als het eerder gekozen punt is, en de iteratie start op $z = 0$
- Itereer de functie, en check of de iteratie richting oneindigheid gaat, of convergeert naar een waarde. Hiervoor zijn soms vele iteraties nodig.
- Als de iteratie richting oneindigheid gaat, hoort de originele c niet bij de set. Als de iteratie convergeert dan hoort de c wel bij de set.
- Plot alle punten in het complexe vlak die bij de verzameling horen, en het resultaat is als volgt:



Ofwel de Mandelbrot Set! Deze render volgt exact het systeem dat hierboven is beschreven. Maar zoals eerder al was te zien in de inleiding, kan de Mandelbrot Set ook worden ingekleurd. Dit gebeurt door middel van een kleuren map, ofwel een instructieboekje voor de computer dat elk punt een speciale kleur geeft aan de hand van de berekeningen van dat punt.
Later zal meer worden besproken over kleurenmappen, en hoe deze gebruikt kunnen worden om een extra dimensie toe te voegen aan een fractal.
Wat opmerkelijk is aan dit soort fractals is het feit dat er nergens een initiële vorm wordt gegeven voor de fractal. In tegenstelling tot een fractal zoals de Sierpinski driehoek, die begint met een gelijkzijdige driehoek. Het enige wat er nodig is om fractals als de Mandelbrot Set te genereren is een grote verzameling aan punten, en een heleboel computerkracht. Het kan in principe met de hand, maar dat zal wel even duren.
Dat brengt een ander punt op over dit soort fractals, namelijk het feit dat naarmate de iteraties op elk punt toeneemt de fractal gedetailleerder zal worden. Dit gaat hand in hand met de “dichtheid” van de berekende punten, ofwel resolutie. Naarmate de resolutie en iteraties op een hoger getal worden berekend zal de uiteindelijke fractal gedetailleerder zijn. Dit impliceert ook dat de fractal meer tijd zal opnemen om te maken.

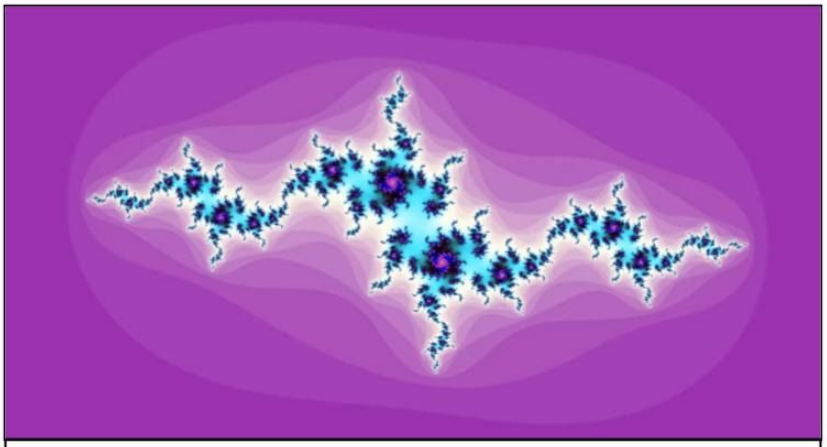
De waardes die zijn gebruikt in het systeem van regels, ofwel Z = 0 als startpunt, en de exponent van Z die gelijk is aan twee, kunnen uiteraard ook veranderd worden. Zoals eerder al naar voren kwam bij de iteratieve functie generatie, heeft dit ook invloed op hoe de uiteindelijke fractal er uit ziet.
Als bijvoorbeeld de formule op Z=1 wordt geïtereerd, komt er dit uit:



De fractal is anders van vorm, maar bevat alsnog soortgelijke eigenschappen van de Mandelbrot Set. Zoals de linker "bobbel", en de dendrieten die ontspringen van deze bobbel.
Indien we de logica van het systeem aanpassen kunnen er nog steeds fractals uit komen.
Bijvoorbeeld:

- Neem een punt uit het complexe vlak, in de vorm: $c = a + bi$
- Deze c wordt, net als bij de Mandelbrot Set, in de formule $F_c(z) = z^2 + c$ gesubstitueerd.
- Neem nog een punt uit het complexe vlak, dit zal *z* worden.
- Itereer de formule voor alle punten uit het complexe vlak, en test nogmaals of deze iteratie richting oneindigheid gaat. Tel het aantal iteraties tot dit gebeurt. Hieruit ontstaat dus een punt, in de vorm van $z = a + bi$ met daaraan het aantal iteraties gekoppeld.
- Indien de iteratie convergeert naar een bepaalde waarde, wordt het aantal iteraties van dit punt op nul gehouden.
- Plot vervolgens alle punten, met een kleur die gebaseerd is op het aantal iteraties van dat punt. En hieruit volgt een fractal in een vorm van:

Dit soort fractals worden Julia Sets genoemd, en er zijn virtueel oneindig soorten sets. De c die benodigd is voor de iteratie kan namelijk elk punt zijn in het complexe vlak. In conclusie, fractals kunnen dus voortkomen uit een systeem van regels, die coördinaten als invoer nemen, en een fractal als uitvoer hebben.



Een Julia Set, gegenereerd met c = 0.4 - 0.325*i
Bron: Jimi Sol *"The Julia Sets: How it works, and why it's amazing"*

Conclusie van het vooronderzoek

De focus van het vooronderzoek lag op het beantwoorden van de deelvragen, en op het aanleggen van een infrastructuur waarmee de multidimensionale fractal kan worden gegenereerd. Voordat het hoofdonderzoek wordt verder gezet, zullen eerst de deelvragen worden beantwoord in de context die tot dusver is onderzocht.

Wat is een fractal, en welke eigenschappen heeft dit wiskundige object?

Een fractal is een wiskundig object dat vormen vertoond die soortgelijk zijn aan het geheel naarmate de schaal veranderd. Fractals hebben ook een intrinsiek patroon dat niet zomaar kan worden beschreven in normale geometrische termen. Een fractal kan ook worden gedefinieerd vanuit een puur wiskundige definitie, die stelt dat de Hausdorff dimensie van de fractal groter moet zijn dan de topologische dimensie, ofwel:

$$F \subseteq \mathbb{R}^n \mid n \in \mathbb{Z},\ dim_H(F) > dim(F)$$

Verder zijn fractals relatief simpel te genereren, en kunnen voortkomen uit een systeem van simpele regels, of functies. Tot slot heeft een fractal een zekere fractale dimensie, ofwel een getal dat impliceert hoe complex het detail van de fractal is naarmate de schaal veranderd.

Hoe wordt een fractal gegenereerd, en hoe kan dit generatie process worden beïnvloed?

Een fractal kan op enorm veel verschillende manieren worden gegenereerd, en komen in allerlei contexten voor, van het modelleren van konijnenpopulaties, tot het itereren van complexe getallen. De drie voornaamste manieren van fractal generatie zijn:

- Iteratieve functie systemen: Hier wordt door een iteratie van een bepaalde regel of manipulatie een fractale vorm gegenereerd, zoals de Sierpinski driehoek.
- Vreemde aantrekkers: Binnen dynamische systemen kunnen fractals voorkomen door de juiste weergave van waardes van het dynamisch systeem, zoals de Feigenbaum Aantrekker.
- Ontsnappingstijd fractals: De methode waar een verzameling van coördinaten wordt geïtereerd, en aan bepaalde condities moet voldoen om deel te zijn van de uiteindelijke fractal. Zoals de Julia Set.

Hoe kan het gebruik van hogere dimensionale datasets worden geoptimaliseerd?

Er zijn een aantal methoden naar voren gekomen om het gebruik van datasets efficiënter te maken. Zoals het gebruik van de Numpy module, die ervoor zorgt dat er met arrays kan worden gewerkt die een stuk sneller te manipuleren zijn. Ook is het potentieel van CUDA besproken, een extensie van Python waardoor de standaard lineaire computatie kan worden omgezet naar parallelle computatie. Desondanks zal het onderwerp van optimaliseren nog steeds worden behandeld in de volgende hoofdstukken. Dit komt voort uit een programmeurs principe, waar vroegtijdige optimalisatie vaak voor problemen kan zorgen naarmate een programma zich ontwikkeld. Dus zal deze vraag nogmaals worden beantwoordt in het hoofdonderzoek.

Het hoofdonderzoek

Inleiding

Zoals besproken in de originele inleiding, zal het voornaamste doel van het hoofdonderzoek zijn om een vierdimensionale Mandelbrot Set te maken. Het onderzoek zal hiervoor verdeeld worden in een aantal subonderwerpen, namelijk hoe de multidimensionale fractal wordt gegenereerd, in beide wiskundige, en computer context.
Het onderzoek zal zich vervolgens richten op hoe de fractal uit vier dimensies kan worden gevisualiseerd in twee of drie dimensies. Ook zal worden besproken hoe de fractal kan worden gevisualiseerd door middel van OpenGL, de rendermethode gekozen in het vooronderzoek.
Daarna zal er nogmaals worden gekeken naar het optimaliseren van Python's gebruik van de datasets, en het renderen daarvan. En daarna zal de hoofdvraag worden beantwoordt, samen met de resterende deelvraag, ofwel:

- Hoe is een multidimensionale fractal te visualiseren in een lagere dimensie?
- Hoe optimaliseren we Python's gebruik van multidimensionale datasets?

Tot slot zal dit onderzoek worden afgesloten met een algemene conclusie, dat alle onderzoeksvragen nog eens concreet beantwoordt. Daarop volgt een discussie over hoe dit onderzoek beter had kunnen worden uitgevoerd, en een dankwoord aan alle mensen die hebben meegeholpen aan dit onderzoek.

Generatie van de multidimensionale Mandelbrot Set

Het genereren van de multidimensionale Mandelbrot Set kan op vele manieren worden aangepakt, die allemaal resulteren in verschillende resultaten. Zo zijn er al een aantal bekende multidimensionale sets, bijvoorbeeld de "Mandelbulb". Deze fractal is geconstrueerd door Daniel White en Paul Nylander door middel van bolcoördinaten, en een formule die het process van de tweedimensionale Mandelbrot Set transleert naar drie dimensies. Deze fractal is dus geconstrueerd in drie dimensies, maar dit onderzoek zal zich bezig houden met de Mandelbrot Set die is geconstrueerd in vier dimensies. Deze fractal komt voort uit een speciaal soort getal, namelijk Quaternions. De werking van deze getallen zal op de volgende pagina worden besproken. Vervolgens zal deze vierdimensionale fractal worden geprojecteerd naar een lagere zichtbare dimensie.

De Mandelbulb
Bron: Ondřej Karlík via Wikipedia

De afwezigheid van driedimensionale complexe getallen

Zoals vermeld in de inleiding is het niet mogelijk om een driedimensionaal complex getal te construeren. Dit komt voort uit de tak van wiskunde die zich bezighoudt met abstracte algebra, waar een zekere heer Frobenius een stelling heeft bewezen die luidt als volgt:

"De enige n-dimensionale associatieve delingalgebras over de reële getallen waar n finiet is, zijn de reële getallen, de complexe getallen, en de quaternions"

Dat is nogal een moeilijke stelling, daarom zal de bewering worden opgedeeld in kleinere delen die uiteindelijk de gehele bewering uitleggen.

Associatieve delingalgebras

Binnen de abstracte algebra is een delingsalgebra de algebra over een lichaam waar deling mogelijk is. Het "lichaam" waar hier over wordt gesproken betekent niets meer dan een algebraïsche structuur waarin kan worden opgeteld, afgetrokken, gedeeld, en vermenigvuldigd. Dit wordt gesteld door Wolfram Mathworld. Voorbeelden van lichamen zijn de reële getallen, of de complexe getallen. Dit in concrete wiskundige notatie ziet er uit als:

Een algebra is een delingsalgebra als:

- De algebra, die niet leeg is:
  - $A \neq \{0\}$
- Voor elke twee elementen:
  - $a,\ b \in A \mid a \neq 0$
- De vergelijkingen:
  - $a * x = b$
  - $y * a = b$
- Beide oplossingen hebben die deel zijn van D zijn:
  - $x,\ y \in A$

Wanneer een algebra associatief is betekent dat het lichaam waarover de algebra strekt ook toestaat dat er distributief en associatief met vectors kan worden vermenigvuldigd. Ofwel:

Een algebra is associatief als:

- De algebra: A, over de vectorruimte: V aan de volgende eisen voldoet.
- Getallen binnen multiplicatie van volgorde kunnen variëren met hetzelfde resultaat:
  - $(x * y) * z = x * (y * z) \mid \{x,\ y,\ z\} \in A$
- De distributie van een x over een som van $y + z$ hetzelfde is als de individuele multiplicatie van x met $y + z$:
  - $x * (y + z) = x * y + x * z \mid \{x,\ y,\ z\} \in A$
- Het product van a in V met $(x * y)$ gelijk is aan het product van $(a * x) * y$ en gelijk is aan $(a * y) * x$:
  - $a * (x * y) = (a * x) * y = (a * y) * x \mid \{x,\ y\} \in A,\ a \in V$

Dus een associatieve delingsalgebra is een algebra over een lichaam, waarin opgeteld, afgetrokken, gedeeld, en vermenigvuldigd kan worden. De vermenigvuldiging staat hier wel onder de eisen die hierboven zijn gesteld.

Het restant van de bewering

Nu de bulk van de bewering is ontcijferd naar een aantal wiskundige eisen, kan er worden gekeken naar de rest van de bewering. Het deel over de "*n-dimensionaliteit* waarin *n* finiet" stelt dat de algebra waarmee wordt gewerkt niet oneindig-dimensionaal mag zijn.
Verder gaat de "*Algebra over de reële getallen*", dit impliceert dat het lichaam van de algebra over de reële getallen strekt.
Dus de bewering stelt dat de enige algebras, die associatief, deelbaar, en finiet-dimensionaal zijn, de reële getallen, de complexe getallen, en de quaternions zijn.
En dit kan anders worden verwoord naar:

"De enige algebras die aan de genoemde eisen hierboven, en in Associatieve delingalgebras voldoen zijn de reële getallen, de complexe getallen, en de quaternions."

Uit deze bewering kan dan ook worden geconcludeerd dat er geen driedimensionale getallen bestaan, tenminste geen getallen die aan de eisen hierboven voldoen. Want de reële getallen zijn eendimensionaal, de complexe getallen zijn tweedimensionaal, en de quaternions zijn vierdimensionaal.
Het bewijs voor deze stelling zal in dit onderzoek afwezig zijn, omdat dit onderzoek zich nou eenmaal bezig houdt met fractale geometrie, en dimensionaliteit. Een bewijs van de stelling kan worden gevonden in de bijlage van dit verslag onder de naam: "*Frobenius Theorem.pdf*".

Hoe de multidimensionale generatie werkt

Nu het duidelijk is waarom er geen driedimensionale getallen staan, kan er worden gekeken naar de vierdimensionale getallen, ofwel de Quaternions. De quaternion getallen zijn voor het eerst beschreven door William Rowan Hamilton in 1843. En waren het resultaat van het proberen te construeren van een driedimensionaal complex getallen systeem, wat dus niet kan volgens de Frobenius stelling, die pas in 1877 werd bewezen.

Quaternions

Een quaternion is opgebouwd uit vier aparte variabelen, waarvan drie een complexe coëfficiënt bevatten:

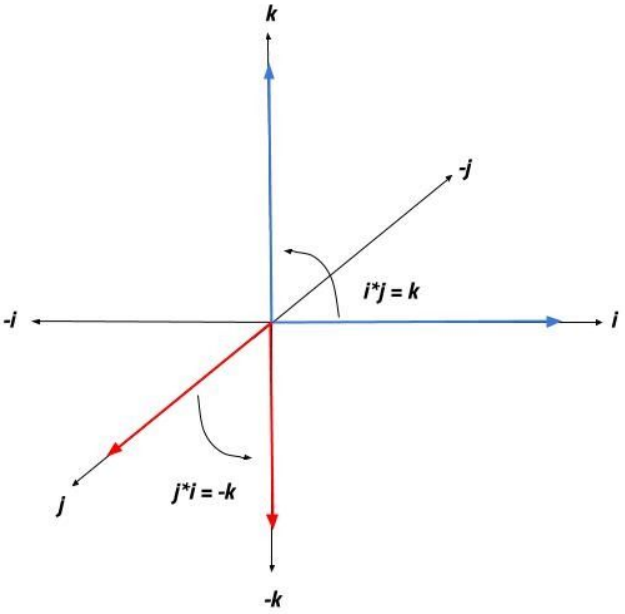
$$Q = a + b * i + c * j + d * k \mid \{a, b, c, d\} \in \Re$$

De complexe coefficenten i, j, k kunnen worden opgevat als eenheidsvectors langs drie ruimtelijke assen. Dit wordt daarom ook het vectordeel van de quaternion genoemd, verder is er nog een reële coëfficiënt, dit wordt het scalardeel genoemd.
Quaternions hebben een bijzondere eigenschap, de getallen zijn namelijk niet commutatief. Dat betekent dat twee quaternions $Q_1 \,\&\, Q_2$, twee verschillende resultaten opleveren wanneer ze worden vermenigvuldigt met elkaar:

$$Q_1 * Q_2 \neq Q_2 * Q_1$$

Een uitleg over waarom dit gebeurt volgt op de volgende pagina.

Deze eigenschap draagt ook over op de complexe coëfficiënten, voor quaternions is $i * j$ dus niet hetzelfde als $j * i$. Deze eigenschap kan geometrisch worden verklaard, en is deel afgeleid van 3Blue1Brown's video over Het visualiseren van Quaternions. In het complexe vlak telt een vermenigvuldiging met i namelijk als een 90° rotatie om de oorsprong. In de quaternion ruimte wordt de rotatie van de coëfficiënt om de bijbehorende as uitgevoerd. Dus een vermenigvuldiging met i impliceert een rotatie rondom de i-as, en een vermenigvuldiging met j om de j-as.



Vanwege deze geometrische eigenschap is de vermenigvuldiging van $i * j$ dus gelijk aan k, want de eenheidsvector i wordt 90° rondom de j-as gedraaid. Dat resulteert in k. Als de vermenigvuldiging van $j * i$ wordt uitgevoerd moet de eenheidsvector van j met 90° worden geroteert rondom de i-as, dat resulteert in $-k$.
De werking van alle coëfficiënten met elkaar kan worden uitgewerkt tot de tabel hiernaast. Als Python met quaternions werkt moet er ook rekening worden gehouden met deze eigenschap.

x	1	i	j	k
1	***1***	***i***	***j***	***k***
i	***i***	***-1***	***k***	***-j***
j	***j***	***-k***	***-1***	***i***
k	***k***	***j***	***-i***	***-1***

De generatie door middel van Quaternions

Quaternions zijn dus een vierdimensionaal getal, met een aantal bijzondere eigenschappen. De tweedimensionale Mandelbrot Set wordt gemaakt door de functie $F_c(z) = z^2 + c$ te itereren op $z = 0$ en c als een punt uit het complex vlak. Indien z wordt gesubstitueerd voor Q, waar Q een quaternion is met $a,\ b,\ c,\ d = 0$ en c substitueren voor een vierdimensionaal punt p uitgedrukt in quaternion vorm. Dan komt de volgende formule aan het licht:

$F_p(Q) = Q^2 + p$

Vervolgens moeten de andere regels nog worden gedefinieerd die ook toegepast zijn op de tweedimensionale set. Deze regels zijn als volgt:

- Als de iteratie op $Q = 0$ richting oneindigheid gaat, dan is dit punt geen deel van de set.
- Als de iteratie op $Q = 0$ convergeert naar een getal, dan wordt dit punt *wel* deel van de set.

Dit is generatieve methode die zal het programma gaan gebruiken, of de verzameling aan getallen die hieruit komt fractale eigenschappen bevat is nog maar de vraag. Maar dat zal uiteraard onderzocht worden. Met de generatieve methode, en de werking van quaternions kan er worden gekeken naar hoe Python deze verzameling aan getallen kan gaan genereren.

Coderen van de generatie

Om te beginnen met het coderen van de fractal dataset generator moet Python eerst begrijpen wat een Quaternion is. Zoals besproken in *"De grootste limitatie van Python",* doet Python aan iets genaamd "type-checking". De interpreter van Python leest dus de codering, en checkt de bijbehorende types die gebruikt zijn in de code. Python heeft een aantal standaard types, namelijk:

- Integers: Hele getallen
- Floats: Decimale getallen
- Complex: Complexe getallen
- Strings: Een sequentie van karakters
- Boolean: Waar of onwaar, ofwel True of False

Het doel is dus om een nieuw type toe te voegen aan Python, dat gelijk staat aan de wiskundige definitie van een Quaternion. Gelukkig zijn quaternions niet onbekend in de programmeurs wereld, en heeft *Michael Boyle*, een module gemaakt voor Python die quaternions als type toevoegt. En dat niet alleen, het type is ook compleet compatibel met Numpy, de module die voor flexibele en snel te manipuleren arrays zorgt. Deze module gecombineerd met Numpy, en standaard Python is genoeg om de dataset te genereren van de vierdimensionale Mandelbrot Set.

De eisen van het programma

Het is noodzakelijk om een aantal eisen van tevoren op te stellen, voordat een programma in elkaar wordt geprogrammeerd. Op deze manier kan er voorkomen worden dat er later stukken codering opnieuw moeten worden geschreven, vanwege een limitatie die pas later opkwam in het programma. De eisen voor het generatieve programma zijn:

- Het programma moet de dataset kunnen genereren volgens de besproken generatieve methode.
- Het programma moet de dataset kunnen opslaan, zodat deze later weer kan worden geladen. Dit zal veel tijd besparen op lange termijn, omdat de dataset dan niet voor elk onderzoek hoeft te worden gegenereerd.
- Het programma moet het aantal punten, en welke punten worden geëvalueerd kunnen veranderen. Dit gaat tegen het "hard-coding" principe in, waar bepaalde waardes en regels handmatig door de programmeur erin worden gezet, en niet door de gebruiker.
- Het aantal iteraties per punt moet kunnen worden aangepast. Vanwege dezelfde redenering als het aantal punten.
- Het programma moet modulair zijn, waardoor het door een ander programma gebruikt kan worden om de dataset te verkrijgen, en zelfs te manipuleren.

Met deze eisen in beeld kan er nu eindelijk worden gekeken naar het programma zelf.

Het generatieve programma

In Python begint een programma altijd met alle modules die moeten worden geïmporteerd:

```
#Imports
import numpy as np
import quaternion
```

Vervolgens moeten alle constanten binnen een functie worden opgezet. Op deze manier van programmeren kan de functie vanuit een ander programma worden opgeroepen, waardoor de eis van modulariteit voldaan is:

```
#Function setup
def dataset_generator():
  #Constant setup
  ONE = np.quaternion(1,0,0,0)
  I = np.quaternion(0,1,0,0)
  J = np.quaternion(0,0,1,0)
  K = np.quaternion(0,0,0,1)
```

De ONE, I, J, K zijn constanten, en zullen dus niet meer worden veranderd in het programma. Daarom zijn ze aangegeven met een hoofdletter.
Vervolgens moeten de variabelen van de functie worden gedefinieerd, zoals eerder benoemd moeten het maximum aantal iteraties, het aantal punten, en welke punten kunnen variëren. Dus:

```
def dataset_generator(corner_1 = [2,2,2,2], corner_2 = [-2,-2,-2,-2],
max_iterations = 256, fineness = 10):
```

“corner_1”, En “corner_2” zijn hier de hoekpunten van een vierdimensionale kubus, die dus met vier coördinaten moeten worden gedefinieerd. De punten die worden geevalueerd zullen zich binnen deze vierdimensionale kubus bevinden. De “fineness” is een getal dat de kubus is kleine stukjes zal verdelen, hogere getallen impliceren een hogere dichtheid van punten, en dus een hogere resolutie fractal. Tot slot is “max_iterations” het maximaal aantal iteraties dat een punt kan doorgaan. De waardes die al zijn gegeven, zijn de standaardwaarden van de functie voor als de functie wordt aangeroepen zonder argumenten.
Vervolgens moet er een Numpy array worden gemaakt met de waardes die moeten worden geïtereerd. Hiervoor zal de *linspace* functie van numpy van pas komen. Deze functie maakt van een domein een array met *n* getallen tussen het domein. Een voorbeeld hiervan:

```
verdeelde_lijn = np.linspace(0, 5, 6, endpoint = True)
print(verdeelde_lijn)
```

Dit resulteert in een array van de vorm:

```
[0. 1. 2. 3. 4. 5.]
```

Indien elke as eerlijk op de *linspace* manier wordt verdeeld, hierna wordt vermenigvuldigd met de bijbehorende coëfficiënt, en vervolgens bij elkaar worden gevoegd ontstaat er een vierdimensionale array die de quaternion ruimte presenteert:

```
  #Four axis setup
  r_axis = np.linspace(corner_1[0], corner_2[0], fineness, endpoint=True)
  i_axis = np.linspace(corner_1[1], corner_2[1], fineness, endpoint=True)
  j_axis = np.linspace(corner_1[2], corner_2[2], fineness, endpoint=True)
  k_axis = np.linspace(corner_1[3], corner_2[3], fineness, endpoint=True)

  #Axis arrangement
  r_axis = real_axis.reshape(fineness, 1, 1, 1)
  i_axis = i_axis.reshape(1, fineness, 1, 1)
  j_axis = j_axis.reshape(1, 1, fineness, 1)
  k_axis = k_axis.reshape(1, 1, 1, fineness)
```

De methode *.reshape(…)* vervormt de array naar de dimensies die worden gegeven in het argument. In het geval van de *r_axis* wordt de array dus vervormt naar een vierdimensionale array, met *fineness* aantal elementen in de eerste dimensie, en een element in alle andere dimensies. Dit staat gelijk aan de reële as in de quaternion ruimte.
Tot slot kunnen de assen worden samengevoegd met hun respectievelijke coefficient:

```
  #Construction of four dimensional quaternion space
  four_d = r_axis + i_axis*I + j_axis*J + k_axis*K
```

Met een *print* statement, en het aanroepen van de functie kan het resultaat tot nu toe worden weergegeven:

```
  ....
  print(four_d)

dataset_generator(fineness = 2)
```

Dit resulteert in:

```
[[[[quaternion(2, 2, 2, 2) quaternion(2, 2, 2, -2)]
   [quaternion(2, 2, -2, 2) quaternion(2, 2, -2, -2)]]

  [[quaternion(2, -2, 2, 2) quaternion(2, -2, 2, -2)]
   [quaternion(2, -2, -2, 2) quaternion(2, -2, -2, -2)]]]


 [[[quaternion(-2, 2, 2, 2) quaternion(-2, 2, 2, -2)]
   [quaternion(-2, 2, -2, 2) quaternion(-2, 2, -2, -2)]]

  [[quaternion(-2, -2, 2, 2) quaternion(-2, -2, 2, -2)]
   [quaternion(-2, -2, -2, 2) quaternion(-2, -2, -2, -2)]]]]
```

Dit resultaat impliceert dat de *four_d* array een vier dimensionale array is. En met een gekozen *fineness* van twee, zijn er zestien punten. En wanneer men deze punten goed bekijkt kan er worden opgemerkt dat dit de hoekpunten zijn van een vierdimensionale kubus. Dat niet alleen, de hoekpunten staan uitgedrukt in *quaternion()* vorm. Dus Python herkent deze punten als een quaternion getal. Dat betekent dat deze array nu kan worden gebruikt voor de iteratieve functie. Het itereren zal gebeuren binnen een loop-bewering, binnen deze loop zal de functie $F_p(Q) = Q^2 + p$ worden geïtereerd. Hiervoor is dus nog een tweede array nodig die Q voorstelt, omdat p staat voor alle punten die moeten worden geëvalueerd. De iteratie van de formule start op $Q = 0$, dus moet er een vierdimensionale array worden gegenereerd die dezelfde vorm heeft als *four_d*, en gevuld is met nullen. Dit kan door middel van Numpy's *zero()* functie, dit geeft een array vol met nullen. Maar in plaats daarvan zal er gebruik worden gemaakt van de *full()* functie, omdat *full()* een array vol maakt met een variabel. Met deze aanpak kan de iteratie dus ook starten op een andere waarde. De *print()* statement zal ook worden verwijderd, omdat deze niet meer nodig is:

```
def dataset_generator(corner_1 = [2,2,2,2], corner_2 = [-2,-2,-2,-2],
max_iterations = 256, fineness = 10, start_value = 0):
...
  #Creation of the "Q-array"
  q_array = np.full((four_d.shape), start_value)
```

Voor deze stap moest er een extra variabel worden toegevoegd aan de originele functie. Zoals hierboven dikgedrukt is weergegeven.

```
  for i in range(max_iterations):
    q_array = np.square(q_array) + four_d
```

Er wordt in de loop gebruik gemaakt van *np.square(q_array)*, dit is een functie binnen Numpy die, zoals verwacht, het kwadraat oplevert van de waardes van de array. De reden voor *np.square(q_array)* is snelheid, in tegenstelling tot de standaard Python manier, ofwel *q_array**2*, berust *np.square()* op C codering die is gecompileerd. En dit maakt de operatie vele malen sneller dan Python, meer hierover is te vinden in het hoofdstuk *"Optimalisatie van het programma"*.
Nu wordt de functie uitgevoerd in een loop uitgevoerd, en deze itereert de vierdimensionale ruimte die is opgezet. Maar nu moet er nog worden gekeken welke punten naar oneindigheid gaan, en welke niet. Daarvoor kan een "*mask*" worden gebruikt, dit is een functionaliteit van Numpy die twee arrays vergelijkt op basis van een logische bewering, en vervolgens het resultaat van de logische bewering uitvoert naar een van de arrays. Om de dataset vervolgens ook te isoleren zal hiervoor nog een array worden gemaakt, deze array kan worden gevuld met nullen:

```
  ...
  #Creation of dataset array
  dataset = np.zeros((as_float_array(four_d.shape)))
  ...
```

Voor de "*mask*" moet nog een logische bewering worden geformuleerd, die checkt of een quaternion naar oneindigheid gaat of niet. Dit is nogal een lastige taak, omdat een quaternion niet groter of kleiner is dan een bepaalde waarde, net als complexe getal. Maar er kan wel worden gekeken naar de lengte van een quaternion ten opzichte van de oorsprong, door middel van vierdimensionale stelling van Pythagoras, ofwel:

$$a^2 + b^2 + c^2 + d^2 = e^2$$

Indien deze formule wordt omgebouwd, en wordt ingevuld met de waardes van de quaternion komt er dit uit:

$$e = \sqrt{a^2 + b^2 + c^2 + d^2} \mid \{a, b, c, d\} \in Q$$

Hierin is e de afstand van de Quaternion tot de oorsprong, en de waarde die onder een bepaalde waarde moet blijven. Deze waarde kan variabel worden gelaten, maar er moet een standaard worden ingesteld. Om dit af te stemmen op de andere standaardwaarde kan er worden gekeken naar het standaard hoekpunt: (2, 2, 2, 2). De afstand van dit standaard hoekpunt tot de oorsprong is gelijk aan:

$$e = \sqrt{2^2 + 2^2 + 2^2 + 2^2} = \sqrt{4 * 2^2} = 4$$

Dus wordt dit variabel weer toegevoegd aan de originele functie, en de "*mask*" kan worden toegevoegd aan de loop:

```
def dataset_generator(corner_1 = [2,2,2,2], corner_2 = [-2,-2,-2,-2],
max_iterations = 256, fineness = 10, start_value = 0, max_distance = 4):
...
  for i in range(max_iterations):
    ...
    dataset[np.abs(q_array) < max_distance] += 1
```

De "*mask*" test of de lengte van een quaternion onder de vier blijft, als dit zo is wordt er aan deze quaternion een "punt" toegekend. Als dit niet zo is wordt de quaternion het "punt" ontkend. Deze bewering deelt de verzameling van getallen die er in wordt gestopt dus op in drie categorieën:

- Quaternions met nul punten, ofwel quaternions die gegarandeerd niet bij de fractal horen.
- Quaternions met punten tussen de een en *max_iterations*. Dit zijn quaternions die op de grens liggen van de set, net als bij de Mandelbrot Set geeft deze grens mogelijkheid voor prachtige visualisaties.
- Quaternions met *max_iterations* punten, ofwel quaternions die wel bij de fractal horen.

Nu de *dataset* array compleet is gegenereerd moet deze, en de *four_d* coördinaten array worden opgeslagen. Hiervoor heeft Numpy een ingebouwde functie, namelijk *np.save()*. Deze functie slaat een array op onder een gegeven naam, ofwel string, en folder op de computer. De string moet alle informatie over de arrays bevatten, anders is het moeilijk om de arrays vervolgens terug te laden voor later gebruik. Hiervoor worden alle variabelen samengenomen voor een getal. Dit getal zal dan de "ID" worden waarmee de arrays weer kunnen worden opgeroepen.

```
  #Strings
  dataset_id = f"{fineness} {max_iterations} {max_distance} {start_value}
{corner_1[0:4]} {corner_2[0:4]}"
  dataset_string = f"dataset_ID={dataset_id}"
  four_d_string = f"four_d_ID={dataset_id}"
```

En tot slot het opslaan van de dataset, met de bijbehorende ID:

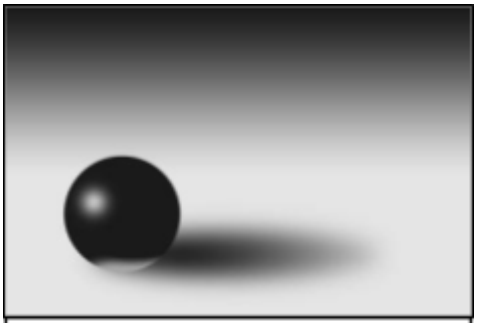
```
#Array saving

np.save(f"E:\\python_projects\\mb_data\\datasets\\{dataset_string}",dataset)
np.save(f"E:\\python_projects\\mb_data\\datasets\\{four_d_string}",four_d)
```

En daarmee is het generatieve programma voor nu af. Het programma genereert een dataset in de vorm van een array met quaternions, die punten in de vierdimensionale ruimte voorstellen. En een array met het aantal iteraties dat deze punten zijn doorgelopen. Nu kan er worden gekeken naar het omzetten van de vierdimensionale quaternions, naar driedimensionale coördinaten. Verder moet Python worden verteld hoe deze punten kunnen worden gevisualiseerd, wat door middel van OpenGL zal worden gedaan.

Projecties van de M.set

De schaduw van een ananas in het zand
Bron: Pineapple Supply Co.

Het idee van een projectie binnen wiskundige context lijkt misschien abstract. Er wordt een hoger dimensionaal punt genomen, daar wordt een manipulatie op gedaan, en er komt een lager dimensionaal punt uit. Maar het idee van een projectie is goed terug te vinden in de realiteit, namelijk in de schaduwen van objecten. Zoals hiernaast te zien is heeft de ananas een schaduw aan de linkerkant van de afbeelding, voor de rest zijn er nog kleinere schaduwen te zien van de hoopjes zand. Uit deze schaduwen weet het menselijk brein de positie van de lichtbron te achterhalen, namelijk aan de rechterkant van de fotograaf, ergens vrij laag aan de horizon. Dit weet het brein te achterhalen doordat de schaduwen relatief lang zijn, en aan de tegengestelde kant van waar de lichtbron zich bevindt. Indien er naar een andere afbeelding wordt gekeken, zoals de afbeelding van de zwarte bal. Weet het brein weer te achterhalen waar de lichtbron is in de afbeelding, ondanks dat de schaduw en het object anders zijn.



Een zwarte bal, met een schaduw aan de rechterkant
Bron: Photokaboom

Dit impliceert twee dingen, er bestaan verschillende soorten schaduwen, en dus projecties van objecten. En het menselijk brein weet het resultaat van de projecties terug te reduceren tot waar de “lichtbron” van de projectie zich bevindt. Voor dit onderzoek zal er voornamelijk worden gekeken naar de verschillende soorten projecties binnen de lineaire algebra en daarbuiten, en hoe deze in Python kunnen worden verwerkt. Omdat in Python deze manipulatie toch numeriek wordt gedaan, kan het concept van de lichtbron achterwege worden gelaten.

Projecties door middel van lineaire algebra

Binnen de wiskundige tak van lineaire algebra wordt er gewerkt met matrices, en vectors. Deze matrices kunnen erg veel dingen betekenen, waaronder een projectie. Gebaseerd op "Getal & Ruimte" en Wikipedia, is een matrix een projectie matrix als:

- $De\ lineaire\ projectie\ P\ over\ een\ vectorruimte\ V\ resulteert\ in\ P^2 = P$

Deze soort projecties kan ook nog worden opgedeeld in orthogonale projecties en oblique projecties. Het verschil tussen deze projecties is dat een orthogonale projectie voldoet aan de volgende eisen:

- De vectorruimte V moet compleet zijn, dit impliceert dat er geen punten missen in de ruimte. Een simpel voorbeeld hiervan is dat de verzameling van rationale getallen $\sqrt{2}$ mist, dus deze is niet compleet.
- De projectie P voldoet aan $P = P^T$

Als een projectie niet voldoet aan deze eisen is het dus een oblique projecties. Maar wat is het verschil tussen deze twee projecties? De orthogonale projectie kan worden gezien als een camera aan de onderkant van een vliegtuig. Dit vliegtuig vliegt over een wijk, en maakt het beeld dat te zien is op de afbeelding:

Orthogonale projectie van een wijk
Bron: eagleview.com

In dit beeld vallen de lijnen die de driedimensionale projecteren loodrecht op de grond, en op de huizen. Orthogonaal is ook een synoniem voor loodrecht, er wordt hier dus gekeken naar een orthogonale projectie. Een eigenschap van orthogonale projecties is dat objecten die parallel waren, parallel zullen blijven in het beeld.

Om het verschil nog beter te laten zien kan er ook worden gekeken naar een oblique projectie van deze zelfde wijk, ofwel de rechterfoto. Hier is te zien dat er een zekere curve zit in alle driedimensionale objecten. Binnen oblique projecties worden parallelle eigenschappen dus niet behouden. Omdat parallelle eigenschappen vervallen valt deze projectie af voor het gebruik voor visualisatie van de multidimensionale fractal. En voor de visualisatie is het een doel om zo veel mogelijk eigenschappen te behouden.
Uiteraard moet er ook worden gekeken naar hoe een projectie er wiskundig uitziet, samen met de integratie in Python.

Oblique projectie van een wijk
Bron: eagleview.com

Lineaire algebra in Python

Om een orthogonale projectie op te stellen die punten uit een vierdimensionale ruimte omzet naar een driedimensionale ruimte, moet er eerst worden gekeken naar een twee dimensionale projectie van drie dimensies. Om dit in wiskundige termen te redeneren:

- $V\ ind\ een\ P,\ die\ \mathbb{R}^3 \rightarrow \mathbb{R}^2$

P is hier een matrix van 3x3, maar omdat de projectie van drie dimensies naar twee gaat zal een van de rijen van de matrix leeg blijven. Voor een driedimensionale projectie van vier dimensies wordt de matrix dus 4x4 met nogmaals een lege rij.
Deze matrix moet ook voldoen aan de eis die is opgesteld in "*Projecties door middel van lineaire algebra*", deze eisen zijn:

- $P^2 = P$
- $P = P^T$
- De vectorruimte V waarvan P projecteert moet compleet zijn.

De laatste eis is al direct voldaan voor de driedimensionale projectie, omdat de vectorruimte hier gelijk staat aan de metrische ruimte waarin de quaternions werken. En de tweedimensionale projectie zal ook in een complete vectorruimte worden geconstrueerd. Verder zijn de eerste twee eisen pas te testen wanneer een matrix wordt opgezet. In Getal & Ruimte VWO D deel 3, hoofdstuk lineaire algebra worden er meerdere tweedimensionale orthogonale projecties opgezet. Een daarvan is relatief simpel:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Het resultaat van van deze projectie met een vector is als volgt:

$$P \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Vervolgens kan er worden gekeken of de eis $P = P^2$ wordt voldaan:

$$P^2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = P$$

Tot slot moet er worden gekeken of de de projectie P voldoet aan $P = P^T$, hierin is P^T de transpositie van de matrix, dat impliceert dat de rijen en kolommen moeten worden omgedraaid:

$$P^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = P$$

Dus kan er worden geconcludeerd dat P een orthogonale projectie is. Vervolgens kan er worden gekeken naar een vierdimensionale orthogonale projectie. Dit kan relatief simpel door de matrix uit te breiden naar een 4x4 matrix, met dezelfde opbouw als de 3x3 matrix, dus:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Deze matrix kan ook worden gezien als de verzameling van eenheidsvectors, waarvan een eenheidsvector is weggelaten. Vervolgens kan er worden getest of de projectie orthogonaal is, en wat voor resultaat de projectie oplevert:

$$P \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} \quad P^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = P \quad P^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = P$$

Deze matrix levert dus ook een orthogonale projectie op, en reduceert een vierdimensionale vector naar een driedimensionale vector. Dat is precies wat nodig is om de vierdimensionale dataset om te zetten naar driedimensionale punten, die vervolgens gerenderd zouden kunnen worden. Het is wel belangrijk om op te merken dat deze projectie een deel van de informatie verloren laat gaan, namelijk het w-coördinaat. Maar als de matrix wordt omgebouwd naar een andere verzameling van eenheidsvectors, bijvoorbeeld de eenheidsvectors van $x,\ y,\ en\ w$. Dan komt dit resultaat uit de projectie:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad P \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ w \end{bmatrix}$$

Door middel van deze projectie kan dus het z-coördinaat van de vierdimensionale vector weg worden gelaten. En dit process werkt uiteraard ook voor de andere coördinaten.

Om Python te laten werken met matrices moet Python weer worden aangeleerd wat een matrix betekent, net als bij de quaternions. Gelukkig heeft de Numpy module al een ingebouwd functie voor matrices, volgens de documentatie. Namelijk *np.matmul(...)*, voor deze functie zijn twee normale Numpy arrays nodig, en die worden vervolgens als matrices vermenigvuldigd.
Als de matrix P dus moet worden gecodeerd komt dat er zo uit te zien:

```
#Imports
import numpy as np

#Matrix P
projection_matrix = np.array[[1, 0, 0, 0],
                             [0, 1, 0, 0],
                             [0, 0, 1, 0],
                             [0, 0, 0, 0]])

#Print result
print(projection_matrix)
```

Met als resultaat:

```
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 0]]
```

Deze matrix kan nu worden gebruikt op een vector, die als array kan worden opgegeven, om het resultaat te krijgen dat eerder is berekend:

```
...
#Vector
four_d_vector = np.array([1,1,1,1])

#Calculation
four_d_vector = np.matmul(projection_matrix, four_d_vector)

print(four_d_vector)
```

Dat resulteert in:

```
[1 1 1 0]
```

Ofwel de x, y, z vector die is berekend op de vorige pagina.
Uiteraard kan de projectiematrix P variëren op vele verschillende manieren. En het programma moet de projectiematrix ook kunnen laten variëren, zodat daarmee kan worden gekeken wat het verschil is tussen verschillende projecties. Een fractal die de z-coördinaat mist vanuit vier dimensies kan er uiteraard anders uitzien dan een fractal die de w-coördinaat mist. Dus is flexibiliteit in het uiteindelijke programma zeer belangrijk.

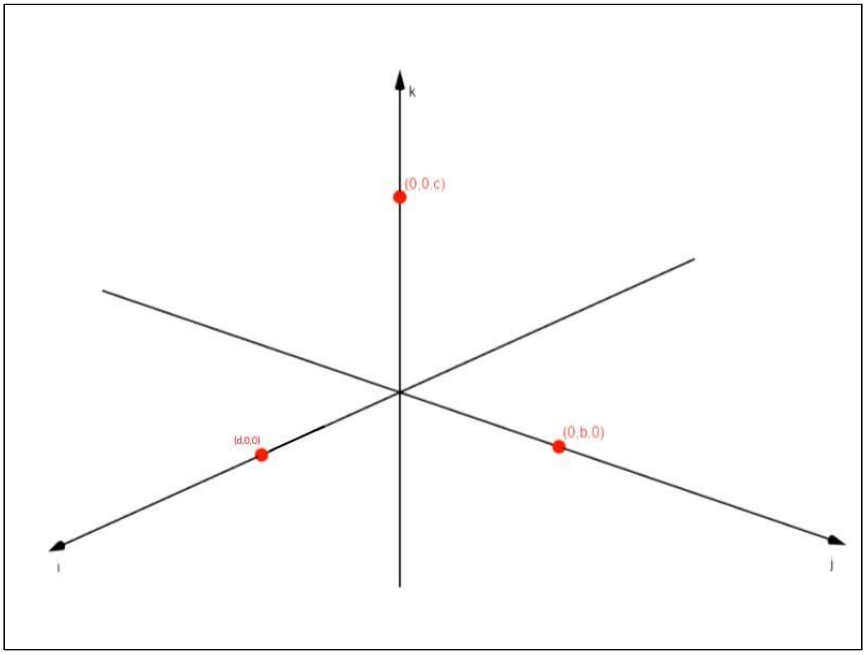
Projecties door middel van geometrie

Matrices werken dus goed als projecties, maar zonder te kijken naar andere mogelijkheden valt er niet uit te sluiten of matrices de beste optie zijn. Daarom zal het concept van projecteren door middel van geometrische transformaties ook onderzocht worden.
Dit proces zal bestaan uit een systeem van regels, en of stappen. Die *n* variabelen als invoer neemt, en een *n-1* variabelen als uitvoer geeft. Dus als er een vierdimensionaal punt wordt ingevoerd komt er een driedimensionaal punt uit. Uiteraard zal er voor deze projectie ook weer worden gekeken hoe het kan worden geïntegreerd in Python, en het uiteindelijke programma.

Om te beginnen aan de geometrische projectie moet er worden gekeken naar de invoer. In de context van dit onderzoek zal dat een vierdimensionaal punt zijn, of een quaternion getal.

$$P_4 = (a, b, c, d) \ \vee \ Q_4 = \mathrm{a}i + \mathrm{b}j + \mathrm{c}k + d$$

De laatste drie van de punt coördinaten kunnen los worden genomen, of het vectordeel van de quaternion kan worden genomen. Dit resulteert in: $b,\ c,\ d$. Vervolgens worden er drie punten geconstrueerd op drie respectievelijke assen in een driedimensionale ruimte. Elk punt zal een van de variabelen als invoer nemen. Dit kan worden gevisualiseerd als volgt:



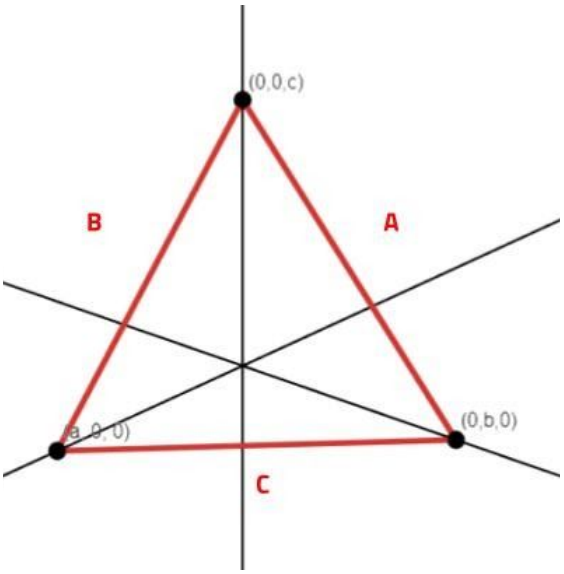
Een scherpe lezer kan al zien waar dit heen gaat, er zal nu namelijk een driehoek worden geconstrueerd. Een driehoek heeft veel geometrische eigenschappen, en deze eigenschappen kunnen worden gebruikt om de projectie te vorderen.
Het construeren van de driehoek kan door middel van vectoren, via deze aanpak kan direct de lengte van elke zijde worden berekend:

$$A = |\vec{b} - \vec{c}|$$

$$B = |\vec{c} - \vec{a}|$$

$$C = |\vec{a} - \vec{b}|$$

De geometrische projectie ziet er nu uit als volgt:
Nu zal er weer een nieuw punt worden geconstrueerd, namelijk het middelpunt van de driehoek. Dit punt komt voort uit het snijpunt van alle bissectrices van alle hoeken van de driehoek. Om de coördinaten van dit punt te kunnen berekenen kan er gebruikt worden gemaakt van de volgende formules:



$$I_X = \frac{\vec{a}_x \cdot A + \vec{b}_x \cdot B + \vec{c}_x \cdot C}{A + B + C}$$

$$I_Y = \frac{\vec{a}_Y \cdot A + \vec{b}_Y \cdot B + \vec{c}_Y \cdot C}{A + B + C}$$

$$I_Z = \frac{\vec{a}_z \cdot A + \vec{b}_z \cdot B + \vec{c}_z \cdot C}{A + B + C}$$

Maar na verdere inspectie kan er worden gezien dat deze formules versimpeld kunnen worden in de huidige context, dat komt doordat alle punten, ofwel de vectors, alleen op de ruimtelijke assen liggen. Dus twee van de componenten van de vector zullen altijd nul zijn. Dit reduceert de formules tot het volgende:

$$I_X = \frac{\vec{a}_x \cdot A + \vec{b}_x \cdot B + \vec{c}_x \cdot C}{A + B + C} \mid \vec{b}_x = 0,\ \vec{c}_x = 0$$

$$I_X = \frac{\vec{a}_x \cdot A}{A + B + C}$$

Vervolgens voor het Y-coördinaat:

$$I_Y = \frac{\vec{a}_y \cdot A + \vec{b}_y \cdot B + \vec{c}_y \cdot C}{A + B + C} \mid \vec{a}_y = 0,\ \vec{c}_y = 0$$

$$I_Y = \frac{\vec{b}_y \cdot B}{A + B + C}$$

En tot slot voor het Z-coördinaat:

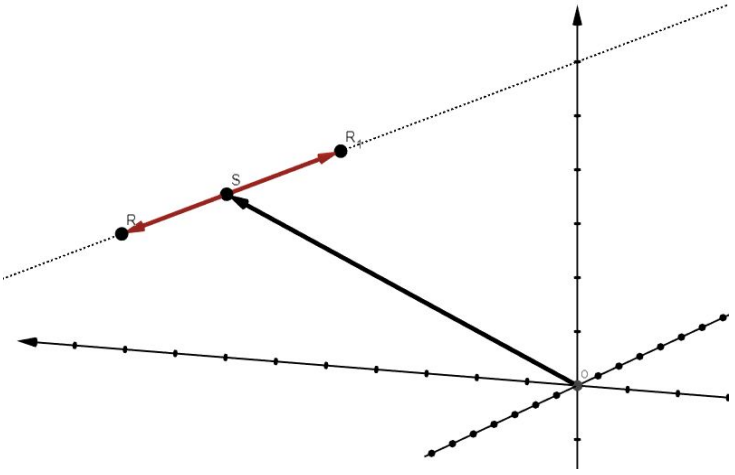
$$I_Z = \frac{\vec{a}_z \cdot A + \vec{b}_z \cdot B + \vec{c}_z \cdot C}{A + B + C} \mid \vec{a}_z = 0,\ \vec{b}_z = 0$$

$$I_Z = \frac{\vec{c}_z \cdot C}{A + B + C}$$

Deze formules resulteren in de coördinaten van het middelpunt van de driehoek. Met dit punt kan een lijn worden geconstrueerd vanuit de oorsprong, deze lijn zal nodig zijn voor de laatste stap. Het construeren van een lijn naar een punt in drie dimensies kan worden gedaan door een vectorvoorstelling. Een vectorvoorstelling is algemeen van vorm als volgt:

$$\vec{L}(\lambda) = \vec{s} + \lambda \cdot \vec{r}$$

Hierin is *s* de zogenaamde 'steunvector', ofwel het punt waar de lijn begint. En *r* is de richtingsvector, die geeft de richting van de lijn aan, en wordt vermenigvuldigt met het variabel lambda. Een visuele representatie van een vectorvoorstelling is hiernaast te zien.



De lijn die zal worden geconstrueerd begint bij de oorsprong, dus de steunvector zal (0,0,0) zijn. En de richtingsvector zal gelijk zijn aan het middelpunt van de driehoek, ofwel punt I. Dat resulteert in deze vectorvoorstelling:

$$\vec{L}(\lambda) = \vec{I} \cdot \lambda$$

Tot slot zal het uiteindelijke driedimensionale punt Q_3 worden geconstrueerd. Dit wordt gedaan door een afstand van variabel *d* over de lijn L te lopen, of in wiskundige taal, het punt Q_3 ligt op afstand *d* van de oorsprong, op de lijn L. Om de coördinaten van dit punt te berekenen zal de volgende vergelijking moeten worden opgelost voor lambda:

$$|\vec{L}(\lambda)| = d$$

Hiervoor moet lambda worden geïsoleerd uit de vectorvoorstelling, gelukkig is dit relatief simpel om te doen. De lengte van de vector berekend met lambda is namelijk proportioneel tot de lengte van de vector keer lambda, ofwel:

$$|\vec{L}| \cdot \lambda = |\vec{L} \cdot \lambda|$$

Met deze identiteit kan de vergelijking worden opgelost:

$$|\vec{L}| \cdot \lambda = d$$

$$\lambda = \frac{d}{|\vec{L}|}$$

Tot slot kan deze waarde van lambda weer terug worden ingevoerd in de vectorvoorstelling, en dit resulteert in de vector, en daarmee coördinaten, van Q_3:

$$\overrightarrow{Q_3} = \vec{L}\left(\frac{d}{|\vec{L}|}\right)$$

Het resulterende beeld van deze projectie kan uiteraard pas worden gezien als de projectie wordt gebruikt door het programma. Dus voor nu is de uitvoer van deze projectie een zwarte doos, maar er kan wel worden gekeken naar hoe deze projectie kan worden geïmplementeerd.

Geometrische projectie in Python

Voordat de geometrische projectie kan worden geprogrammeerd moet er eerst worden gekeken naar de data die al is gegenereerd vanuit de dataset generator. Deze data komt in de vorm van een numpy array, en heeft meerdere dimensies. Dus als er moeten worden gewerkt met de data van deze array moet er wel goed worden gespecificeerd hoe deze data behandeld wordt.
De data die uit de dataset generator komt heeft een array vorm van:

```
(fineness, fineness, fineness, fineness)
```

Hierin is '*fineness*' het variabel in de dataset generator dat de afstand tussen de punten vergroot of verkleint. Een hogere *fineness* impliceert een grotere verzameling aan punten, die dichter bij elkaar liggen. Deze array werkt ook nog met het quaternion datatype, en dat is op zich geen probleem. Maar het is makkelijker als de data in array vorm is. Daarvoor kan de functie *as_float_array()* van de quaternion module worden gebruikt. Dit verandert wel de vorm van de array naar:

```
(fineness, fineness, fineness, fineness, 4)
```

De vier komt voort uit het feit dat het quaternion datatype wordt omgezet naar een kleine eendimensionale array van vier decimale getallen. Hieronder is de functie *as_float_array()* in actie te zien:

```
import numpy as np
from quaternion import *

quaternion = quaternion(1,1,1,1)
print("Quaternion vorm:")
print(quaternion)

print("Array vorm:")
print(as_float_array(quaternion))
```

Met als resultaat:

```
Quaternion vorm:
quaternion(1, 1, 1, 1)
Array vorm:
[1. 1. 1. 1.]
```

Nu de vorm van de data is achterhaald kan er worden gekeken naar hoe de geometrische projectie kan worden geprogrammeerd. Uiteraard begint het programma met de 'imports', ofwel welke modules er gebruikt zullen worden:

```
#Imports
Import numpy as np
From quaternion import *
```

Het is makkelijk in gebruik om de projectie als functie op te zetten, dus:

```
def geometric_projection(input_quaternion):
    '''Projects a 4D quaternion to a 3D point'''
```

Om het rekenwerk te beginnen moet alle variabelen worden geïsoleerd uit de quaternion data, en daarvoor moet de ingevoerde data worden omgezet naar een decimale array. Het kan ook geen kwaad om te testen of de array daadwerkelijk in quaternion vorm is, zodat er geen fouten worden gemaakt:

```
    if type(input_quaternion) == quaternion:
        input_quaternion = quaternion.as_float_array(input_quaternion)

    #Put every coefficient of the quaternions in seperate arrays
    a = input_quaternion*np.array([1,0,0,0])
    b = input_quaternion*np.array([0,1,0,0])
    c = input_quaternion*np.array([0,0,1,0])
    d = input_quaternion*np.array([0,0,0,1])
```

Om de variabelen te isoleren wordt de ingevoerde array vermenigvuldigt met een array die de andere variabelen naar nul zet. Hierdoor is de functie relatief flexibel met welke data mag worden ingevoerd. Deze aanpak impliceert wel dat de uiteindelijke arrays moeten worden opgeschoond, omdat ze vol zitten met nullen. Daarvoor kan de '*mask*' functionaliteit van Numpy weer worden gebruikt:

```
    #Remove all zeros from the coefficient arrays
    a = a[a != 0]
    b = b[b != 0]
    c = c[c != 0]
    d = d[d != 0]
```

Deze isolatie resulteert in vier verschillende arrays die elk een apart variabel bij zich dragen. Dus de array *a* draagt alle *a* variabelen van de originele quaternions die zijn ingevoerd.
Met deze arrays kan er worden gerekend, om te beginnen zullen de lengtes van de zijdes van de driehoek worden berekend:

```
    #Projected triangle edges
    edge_bc = np.sqrt(b**2 + c**2)
    edge_cd = np.sqrt(c**2 + d**2)
    edge_db = np.sqrt(d**2 + b**2)
```

Vervolgens wordt de summatie van de zijdes berekend, ofwel de '*A* + *B* + *C*' die onder de deelstreep staat in de formules. Dit wordt apart berekend zodat de codering beter leesbaar is:

```
    #Edge summation
    s = edge_bc + edge_cd + edge_db
```

Daarna kunnen de coördinaten van het middelpunt van de driehoek worden berekend, volgens de formules die eerder zijn achterhaald:

```
    #Incenter coordinates of triangle
    in_x = (edge_bc * b) / s
    in_y = (edge_cd * c) / s
    in_z = (edge_db * d) / s
```

Dat wordt vervolgd met de lengte van het middelpunt tot de oorsprong:

```
    #Length from origin to incenter
    incenter_length = np.sqrt(in_x**2 + in_y**2 + in_z**2)
```

En daarmee zijn alle nodige variabelen voor de uiteindelijke coördinaten berekend, die tot slot kunnen worden berekend:

```
    #Determined length of line formula
    q_3_x = (a / incenter_length) * in_x
    q_3_y = (a / incenter_length) * in_y
    q_3_z = (a / incenter_length) * in_z
```

Dit resulteert in drie arrays, de eerste array houdt alle x-coördinaten, de tweede de y-coördinaten, en de derde de z-coördinaten. Maar het doel is natuurlijk om een punt uit de projectie te krijgen, dus de arrays moeten nog bij elkaar worden gevoegd. Maar dit kan niet zomaar, als de arrays aan elkaar worden geplakt dan resulteert dit in het door elkaar halen van de coördinaten. Daarom moet elke waarde van de eerste array worden gekoppeld met de corresponderende waarde van de tweede en derde array. Dit kan gelukkig makkelijk worden gedaan door middel van numpy's *stack*() functie:

```
    print("Stitching coordinate arrays...")
    points = np.stack((q_3_x, q_3_y, q_3_z), axis = 1)
```

Tot slot moet de *points* array nog worden teruggestuurd vanuit de functie, dus:

```
    return points
```

En hiermee is de geometrische projectie klaar, een kleine test via deze setup:

```
quaternion_array = np.array([np.quaternion(1,1,1,1),
np.quaternion(-1,-1,-1,-1)])
points = geometric_projection(quaternion_array)
print(points)
```

Geeft dit als resultaat:

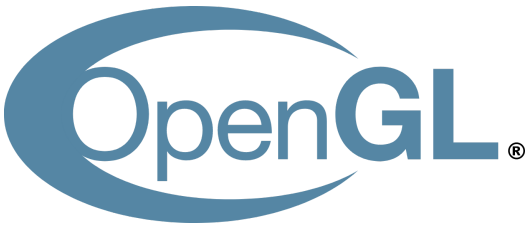
```
[[0.57735027 0.57735027 0.57735027]
 [0.57735027 0.57735027 0.57735027]]
```

Conclusie over projecties

In de afgelopen twee hoofdstukken is besproken hoe punten uit hogere dimensies naar lagere dimensies kunnen worden gehaald door middel van een variatie aan methoden. Waaronder het gebruik van matrices, die verschillende soorten projecties kunnen opleveren afhankelijk van het soort matrix dat wordt gebruikt. Dat niet alleen, ook simpele geometrische stappen kunnen een multidimensionaal punt projecteren naar een zichtbare dimensie. Beide methoden zijn verwerkt in Python, en kunnen worden gebruikt om de multidimensionale dataset om te zetten naar een driedimensionale dataset. Met deze info kan er worden gekeken naar de visualisatie van de vierdimensionale Mandelbrot Set via OpenGL en Python.

Visualisatie van de set

Het visualiseren van de fractal zal door middel van OpenGL en Python gebeuren. De dataset die is gegenereerd in "*Het generatieve programma*" zal hiervoor worden gebruikt. Er zit een groot aantal stappen tussen het genereren van de dataset en het visualiseren door middel van OpenGL. Dit komt doordat OpenGL vrij 'diep' binnen de computer werkt, er wordt bijvoorbeeld gebruik gemaakt van zogenaamde '*Vertex Attribute Objects*'. Dit zijn stukken informatie die OpenGL vertellen hoe stukken geheugen moeten worden geïnterpreteerd. Dit is heel anders dan Python, waar het meeste van dit soort instructies achter de schermen worden opgesteld, en worden geïnterpreteerd vanuit de Python codering. Dit impliceert dus een zekere gelimiteerde aard van Python, of een enorme vrijheid van OpenGL, afhankelijk van het perspectief.



Om een goed beeld te krijgen van hoe OpenGL werkt zullen alle globale stappen worden besproken, en worden geprogrammeerd. Daarentegen zullen de gedetailleerde eisen specifiek aan dit onderzoek, en programma niet uitbundig worden besproken, omdat hiervoor achtergrondinformatie nodig is die niet relevant is tot dit onderzoek.
Er zal wel documentatie over de diepere werking van OpenGL worden toegevoegd aan de literatuurlijst van dit onderzoek, indien een lezer hier behoefte aan heeft kan hij of zij deze documentatie bekijken voor een beter begrip van OpenGL. Met deze disclaimer uit de weg, kan er worden begonnen aan het uiteindelijke programma, en de visualisatie van de multidimensionale fractal.

Het gebruik van OpenGL en de globale werking

OpenGL staat voor Open Graphics Library, het is dus een bibliotheek van allerlei functies die kunnen worden gebruikt om computer graphics te coderen. Voordat OpenGL gebruikt kan worden is er wel wat voorbereiding nodig. Om dit hoofdstuk overzichtelijk te houden zal daarom een tweede inhoudsopgave worden beschreven, die gelijk loopt met het programma dat de uiteindelijke beelden zal renderen. In deze structuur zal ook kort worden besproken wat elk onderdeel doet.

- **Programma**
    - Het hoofdprogramma zal alle relevante onderdelen bevatten. En de benodigde imports en variabelen.
    - **Dataset Generator**
        - De generator maakt de dataset volgens de besproken methode van generatie van de multidimensionale Mandelbrot Set.
    - **Opzet van een glfw 'window'**
        - OpenGL heeft een scherm nodig om in te kunnen renderen, dit scherm moet worden aangemaakt, en juist worden ingesteld. Dit gebeurt via glfw, ofwel het OpenGL Framework.
    - **Shaders**
        - Shaders zijn programma's die naar de videokaart worden gestuurd. De shaders zijn geschreven in GLSL, een andere codering die lijkt op C++. Shaders hebben verschillende functies.
        - **Vertex Shader**
            - De Vertex Shader zorgt ervoor dat data op de juiste plek op het scherm komt. De Vertex Shader zorgt ook voor perspectief in de scene, waardoor er '3D' kan worden gerenderd op een 2D beeldscherm.
        - **Fragment Shader**
            - De Fragment Shader regelt alles betreffende kleur.
    - **Shader Programma**
        - De shaders moeten als een programma naar de kernen van de videokaart worden gestuurd, dit wordt ook wel 'linken' genoemd. Dit zorgt ervoor dat elke individuele kern op de videokaart de juiste instructie volgt, zoals opgegeven in de shaders.
    - **Opzet van OpenGL data**
        - OpenGL kan niet zomaar data gebruiken, de data moet eerst in nette pakketjes worden gestopt, met instructies over hoe deze pakketjes moeten worden geïnterpreteerd. Dit gebeurt via VBO's en VAO's.
    - **De 'render' functie**
        - Tot slot moet OpenGL een signaal krijgen dat het renderen kan beginnen. Dit gebeurt binnen een loop, zodat het beeld continu is. In deze loop kan ook worden gecheckt voor invoer van de gebruiker.
    - **De 'main' functie**
        - Hier worden alle stappen zoals hierboven beschreven uitgevoerd, dit is ook de functie die wordt aangeroepen wanneer het programma wordt uitgevoerd.

Het hoofdprogramma

Om te beginnen met het programma zullen alle nodige modules moeten worden geïmporteerd. Per geïmporteerde module zal worden beschreven wat het nut is van de module binnen het programma:

```
#Imports
import numpy as np
import quaternion

from math import sqrt
from ctypes import c_void_p, c_float, c_uint, sizeof

from OpenGL.GL import *
import glfw
import glm
```

Numpy is uiteraard nodig geweest voor de generatie van de dataset. Numpy is, zoals eerder vermeld, een geoptimaliseerde module voor het verwerken van veel data. Numpy werkt in combinatie met de Quaternion module, deze module voegt een nieuw datatype toe dat een quaternion representeert.
De import van 'math' is de wortelfunctie, *sqrt()*. Hier valt niet veel over te zeggen behalve dat de ingebouwde vorm van wortels in Python, ofwel `'a**(1/2)'`, niet altijd even toepasselijk is.
Ctypes is een module die Python laat werken met c-typen van data. Dit is nodig geweest voor OpenGL, omdat OpenGL is geschreven in C. En niet elke functie van OpenGL is even goed getransleerd naar Python. Daarom was het soms nodig om direct met C types te werken.
Tot slot wordt alles van de OpenGL. GL module geimporteerd, samen met glfw, en glm. De OpenGL.GL module biedt alle functionaliteit van OpenGL aan in Python. Glfw staat voor OpenGL FrameWork, en zorgt voor functies die schermen kunnen maken met een OpenGL context. GLM staat over OpenGL Mathematics, en zorgt uiteraard voor wiskundige operaties die goed aansluiten op OpenGL, zoals het construeren van matrices, vectors, en meer.
Vervolgens kan het volgende stukje codering worden toegevoegd:

```
#Run main function when file is run
if __name__ == '__main__':
    main()
```

Dit zorgt ervoor dat de functie *main()* wordt uitgevoerd wanneer het gehele programma wordt uitgevoerd. Alle relevante onderdelen moeten dus deel worden van *main()*, of een functie die *main()* aanroept.

```
def main():
      '''The main function of the program'''

      #Place your functions here...
```

Dataset Generator

De dataset generator die is beschreven in *Het generatieve programma* kan worden verplaatst naar het hoofdprogramma. Dit is niet helemaal volgens protocol, omdat het limiterend is op lange termijn. Maar dit programma wordt alleen door de auteur, en potentieel lezers van dit verslag gebruikt. Daarom zal het protocol worden gebroken, en alle functies in een document worden verzameld.

```
def dataset_generator(corner_1 = [2,2,2,2], corner_2 = [-2,-2,-2,-2],
max_iterations = 256, fineness = 10, start_value = 0, max_distance = 4,
save_mode = False, filter_mode = False):
```

Een scherpe lezer merkt misschien op dat er een aantal variabelen bij zijn gekomen, namelijk *save_mode* en *filter_mode*. Deze variabelen vertellen de functie of de gegenereerde data moet worden opgeslagen, en of de data gefilterd moet worden. Het systeem van opslaan werkt als volgt:

```
if save == True:
      #String Setup
      ...
      If filter_mode == True:
            dataset_string = f"filtered_dataset_ID={dataset_id}"
      #Saving
      ...
```

Dit systeem zorgt voor meer flexibiliteit in het opslaan van data. De filter is toegevoegd om alleen nuttige data uit de gegenereerde datasets te halen. “Nuttige” data wordt hier geïnterpreteerd als alle punten die het maximum aantal iteraties heeft doorstaan, ofwel alle punten die deel zijn van de Mandelbrot Set. De filter methode is nogal onorthodox, maar werkt zeer effectief:

```
if filter_mode == True:
    print("Starting filtering...")
    four_d = quaternion.as_float_array(four_d)
```

Eerst wordt er getest of de filter aan staat, daarna wordt de *four_d* array omgezet naar een array met decimalen.

```
    #Masks all points which did not make it to max_iter with -4.20
    four_d[dataset != max_iterations] = -4.20
    dataset[dataset != max_iterations] = -4.20
```

Vervolgens wordt door middel van een *‘mask’* alle punten die niet het maximum aantal iteraties heeft bereikt vervangen door de waarde: -4.20. Dit gebeurt ook voor de array die alle iteratie waarden vasthoudt. De waarde hier had in feite elk negatief getal kunnen zijn, omdat *four_d* wordt gemaskeerd tegenover *dataset*. Vervolgens gebeurt dit ook bij *dataset* zelf, en omdat deze array geen negatieve getallen kan bevatten, is een negatief getal een goede waarde om makkelijk te kunnen filteren.

Vervolgens worden de twee array gecombineerd tot een grote array, dit is om de data klaar te maken voor de OpenGL data opzet. Ook wordt de grootte van de array opgeslagen, om te kunnen gebruiken als vergelijkingsmateriaal:

```
    #Combines the two masked sets
    combined_dataset = np.concatenate((four_d, dataset), axis = 4)
    initial_size = combined_dataset.size
```

Daarna worden alle waarden van -4.20 uit de gecombineerde dataset gefilterd, en wordt de nieuwe grootte van de array vastgesteld:

```
    #Filters out all values of -4.20
    combined_dataset = combined_dataset[combined_dataset != -4.20]
    new_size = combined_dataset.size
```

Tot slot worden de twee waardes mooi uitgevoerd naar de terminal:

```
    #Feedback
    print(f"Filtering succeeded!\nReduced size from {initial_size} to
{new_size}\n")
```

Het installeren van deze filter is wat vroegtijdige , maar nodige optimalisatie. Het effect van de filter is namelijk zeer groot. Helemaal bij groot genoege datasets, bijvoorbeeld een dataset van [-2, -2, -2, -2] tot [2, 2, 2, 2], met een *fineness* van 50:

```
Reduced size from 46118408 to 183240
```

Dat is een reductie van 99.6% van data die moet worden opgeslagen. Een zeer nuttige vroegtijdige optimalisatie dus.
Uiteraard moet de functie ook nog werken zonder filter, dus:

```
    if filter_mode == False:
        #Combines the four_d array and iteration dataset array
        four_d = quaternion.as_float_array(four_d)
        combined_dataset = np.concatenate((four_d, dataset), axis = 4)
```

Hier wordt de *four_d* array omgezet naar decimalen, en dan worden de twee arrays weer samengevoegd tot een grote array. Als voorbereiding op de OpenGL data opzet.

De opzet van een glfw 'window'

OpenGL heeft een stukje scherm nodig waar het kan beginnen met renderen, dit stukje scherm moet wel worden aangewezen voor OpenGL, anders gebeurt er niks. Voor het aanmaken van zo'n scherm kan er gebruik worden gemaakt van het OpenGL FrameWork. Dit begint met het aanmaken van de functie, en de nodige variabelen, namelijk de hoogte en breedte van het scherm:

```
def make_window(width = 1920, height = 1080):
    '''Makes an OpenGL contextual window, where objects can be rendered'''
```

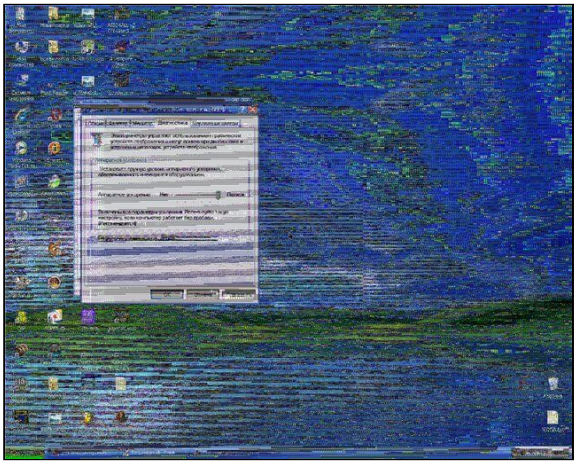
Daarna moet de glfw bibliotheek worden opgestart, dit gebeurt door middel van de glfw.*init()* methode. En uiteraard wordt er wat feedback naar de terminal gestuurd:

```
    #Enables the glfw library, and initializes it
    print("Initiliasing GLFW...")
    if not glfw.init():
        return
    print("Initialized GLFW")
```

Daarna moet er aan glfw worden verteld wat er allemaal met het scherm moet gebeuren. Dit gebeurt door middel van 'window hints':

```
    #glfw Setup
    glfw.window_hint(glfw.CONTEXT_VERSION_MAJOR, 4)
    glfw.window_hint(glfw.CONTEXT_VERSION_MINOR, 3)
    glfw.window_hint(glfw.OPENGL_PROFILE, glfw.OPENGL_CORE_PROFILE)
    glfw.window_hint(glfw.DOUBLEBUFFER, 1)
```

De eerste twee hints vertellen glfw dat OpenGL versie 3, tot versie 4 gebruikt mag worden. De derde hint geeft aan dat er wordt gewerkt met de 'OpenGL core profile', dit is een van de twee opties die OpenGL biedt. De andere optie is 'immediate mode', in deze versie kan er direct worden gerenderd, maar zonder veel van de functionaliteit die OpenGL biedt. Daarom is er gekozen voor het 'core profile'. Tot slot wordt er aan glfw verteld dat er wordt gewerkt met een dubbele buffer.
Een dubbele buffer impliceert dat er per beeld eigenlijk twee beelden worden gegenereerd, indien dit niet wordt gedaan is er meestal sprake van zogenaamde artefacten. Ofwel stukken van het beeld die zijn geweest, maar niet meer actief horen te zijn. Een voorbeeld is hiernaast te zien.



Vervolgens kan het scherm worden aangemaakt, dit gebeurt door middel van de *create_window()* functie:

```
    #Creates a window
    window = glfw.create_window(width, height, "M.Set Generator", None)
```

Vervolgens moet het gecreëerde scherm worden geupdate met OpenGL context, en daarna kan ook worden ingesteld hoe groot de *viewport* moet zijn. De *viewport* is het 'raam' waar de gebruiker doorheen kijkt, en wat OpenGL dus allemaal rendert. Tot slot wordt het scherm, en een lijst van hoogte en breedte teruggevoerd aan de originele functie oproep. Daarmee is de opzet van het scherm klaar:

```
    glfw.make_context_current(window)
    glViewport(0, 0, width, height)
    #Display list
    display = (width, height)
    return window, display
```

Shaders

Deel van OpenGL werkt door middel van iets genaamd 'shaders'. Dit zijn programma's geschreven in GLSL, ofwel OpenGL Shading Language, die naar de videokaart worden gestuurd. In een videokaart zitten een heleboel parallelle eenheden die elk een calculatie kunnen uitvoeren, in tegenstelling tot de CPU, waar een aantal grote kernen in zitten die elk enorm snel calculaties kunnen uitvoeren. Dit process van paralelle computatie presteert een heel stuk beter dan lineaire computatie wanneer er wordt gewerkt met graphics. Dit komt doordat het renderen van geometrie vaak bestaat uit een heleboel simpele calculaties, en niet perse complexe calculaties. OpenGL werkt met een aantal verschillende shaders, elk met een verschillende functie. Maar om te beginnen met het renderen van een simpele scene hoeft er alleen een Vertex Shader, en Fragment Shader worden geprogrammeerd.
Om de functie van shaders beter in perspectief te brengen zullen de stappen van OpenGL worden behandeld, vervolgens wordt in de individuele hoofdstukken van de Vertex Shader en Fragment Shader de shader geprogrammeerd.

De stappen van de OpenGL render procedure

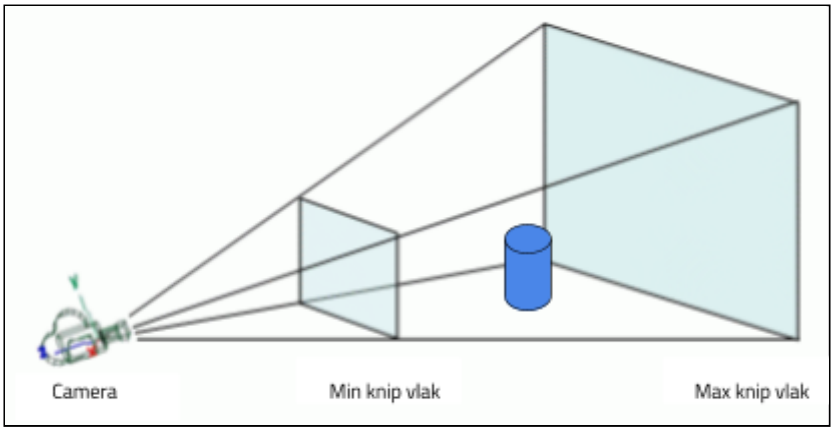
Het renderen van een scène begint in OpenGL bij de data. Deze data wordt aangevoerd door speciale blokken aan geheugen. Hoe deze data wordt opgeslagen en gelezen zal worden behandeld in *Opzet van OpenGL data*. Voor nu kan er worden aangenomen dat er drie getallen, die een positie in drie dimensies representeren, naar de videokaart worden gestuurd.
Vervolgens wordt deze data in de Vertex Shader gevoerd, hier worden de vier getallen omgezet naar de juiste locatie op het scherm. Maar dit is niet zo simpel als het lijkt, een beeldscherm is namelijk tweedimensionaal, en de scène die moet worden gerenderd is driedimensionaal. Dus moet er iets met de driedimensionale ruimte gebeuren waardoor de scène juist wordt weergegeven op het scherm. Hiervoor worden matrices gebruikt, er zijn minimaal drie matrices nodig. Elke matrix heeft een eigen functie, namelijk:

De 'model matrix'
Deze matrix zet door middel van matrixvermenigvuldiging punten om naar een 'gelokaliseerde' ruimte. Dat houdt in dat alle ingevoerde data wordt omgezet naar getallen die deel zijn van het coördinatensysteem van de scène. Dit wordt gedaan om later perspectief toe te voegen. Het is ook mogelijk om via de model matrix ingevoerde data te transleren, en roteren.

De 'perspectief matrix'
De perspectief matrix zorgt ervoor dat de lokale ruimte wordt omgezet naar een scène met perspectief. Dit gaat door middel van projectie die de wereld binnen een frustum projecteert, zoals te zien in de afbeelding. Dit impliceert dat als een object zich verder van de camera beweegt, het ook daadwerkelijk kleiner wordt. Net zoals dat in het echt gebeurt.

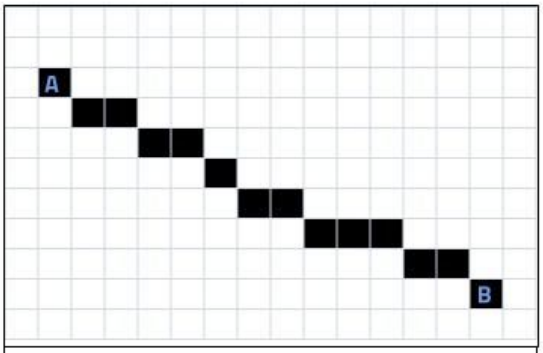
De perspectief matrix is afhankelijk van een aantal variabelen, zoals het gezichtsveld, aspect ratio van het scherm, en de zogenaamde 'knip vlakken'.

- Het gezichtsveld is gelijk aan de hoek tussen de rechter- en linkerkant van de camera. Alles hiertussen wordt gerenderd.
- Het aspect ratio is het ratio tussen de lengte en breedte van het scherm. Dit is benodigd om het perspectief te corrigeren aan de hand van de grootte van het scherm.
- Tot slot zijn de 'knip vlakken' zijn twee waardes. De eerste waarde is gelijk aan de minimale toegestane afstand tussen de camera en de objecten van de scène. De tweede waarde is gelijk aan de maximaal toegestane afstand, ofwel hoe ver de camera maximaal kan zien. Dat impliceert gelijk ook tot hoever OpenGL objecten in de scène rendert.



De 'projectie matrix'

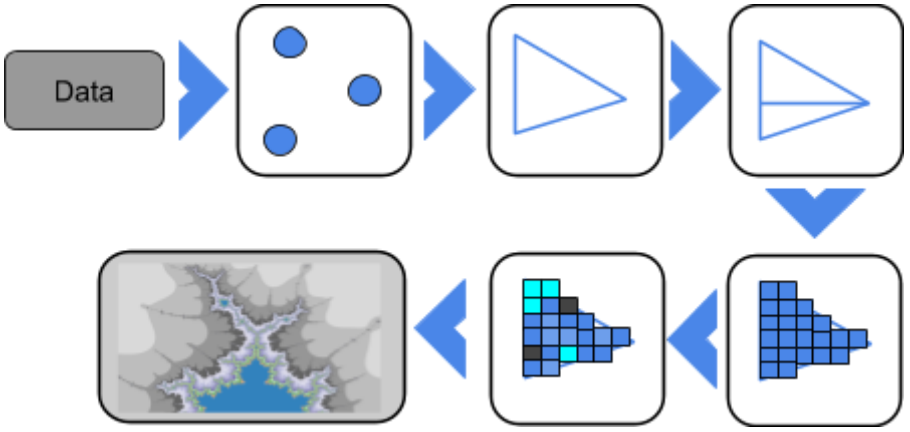
Tot slot zet de projectiematrix de gehele scène om naar genormaliseerde coördinaten. Dat betekent dat alle coördinaten tussen het bereik van -1 en 1 worden geprojecteerd. Indien al deze matrices correct zijn geconfigureerd zal de scène driedimensionaal worden gerenderd.
De Vertex Shader heeft zijn taak dan voltooid, en dan wordt de data van de locatie van alle pixel naar de volgende stage van OpenGL gestuurd, namelijk de geometrische shader. Hier worden 'primitieven' berekend en geconstrueerd, primitieven binnen OpenGL zijn geometrische vormen, zoals punten, lijnen, driehoeken, etc. Als er bijvoorbeeld een vierkant moet worden gemaakt van vier punten in driedimensionale ruimte, dan kan de geometrie shader daarvoor zorgen. Bijvoorbeeld door twee driehoeken te tekenen tussen de vier punten. De uitvoer van de geometrie shader wordt vervolgens gerasterd, ofwel verandert naar pixels data. Als er bijvoorbeeld een schuine lijn moet worden gerenderd, dan moet OpenGL deze schuine lijn weergeven over een raster van vierkante pixels.



Een pixel-lijn van A tot B.

Ditzelfde proces geldt voor driehoeken, en alle vormen die daarvan gemaakt kunnen worden. Deze data wordt als 'fragmenten' doorgegeven aan de Fragment Shader.

De Fragment Shader kleurt vervolgens elk van deze 'fragmenten' in met een opgegeven kleur. Natuurlijk is er meer mogelijk dan alleen een kleur opgeven, binnen de Fragment Shader is het ook mogelijk om licht te simuleren, en daarmee ook schaduwen. Tot slot start de laatste fase van het process, en dit heeft niet een specifieke functie. In de laatste fase kan een grote variatie aan transformaties worden uitgevoerd, reikend van kleuren mengen, tot het toevoegen van hele nieuwe objecten gebaseerd op de eerder verkregen posities uit de Vertex Shader. De uitvoer van deze laatste fase is in de vorm van pixeldata die het beeldscherm kan begrijpen. En daarmee is het renderproces klaar, alhoewel het programmeren van dit alles wel iets gecompliceerder is dan hierboven staat vermeld. Ondanks dat kan er nu wel worden gekeken naar de Vertex en Fragment Shader.



Vertex Shader

Om de codering van de Vertex Shader naar OpenGL te sturen moet het in een string worden geschreven. OpenGL interpreteert deze string met codering dan als GLSL, ofwel de taal waar shaders mee werken. Elk shader programma begint met een versie declaratie, dit vertelt de shader welke versie van OpenGL gebruikt moet worden. Dus:

```
position_vertex_shader_string = '''
#version 330 core
```

Vervolgens moet er vanuit het geheugen van de videokaart data worden gehaald, dit gebeurt door middel van een '*layout call*'. In de layout call moet ook een index getal worden gespecificeerd, de index van de data zal uiteindelijk dus gelijk moeten worden gesteld aan het getal dat in de shader wordt opgegeven:

```
layout (location = 0) in vec4 coordinate;
```

De data die wordt ingevoerd wordt opgeslagen als een vector met vier componenten, onder de naam coordinate. Vervolgens moeten de matrices worden ingevoerd, dit gebeurt door middel van een 'uniform'. Dit is anders van de invoer van data uit het videokaartgeheugen, omdat uniformen globaal worden gebruikt door elke shader, en door elke individuele kern die de shader uitvoert:

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
```

Met alle benodigde variabelen kan de daadwerkelijke functie worden geprogrammeerd, dit gebeurt binnen een *main()* functie:

```
void main(){
//functions here... }
```

De Vertex Shader heeft zijn taak volbracht als de locatie van een stukje data is bepaald. Om deze locatie door te geven wordt het vastgestelde variabel gl_Position gebruikt. GLSL heeft ook een functionaliteit genaamd 'swizzling', deze functionaliteit staat de programmeur toe om componenten van een vector te pakken, en als individuele waarden te gebruiken. Hiermee kan dus een projectie worden geprogrammeerd zonder een matrix te gebruiken, dit gaat als volgt:

```
    gl_Position = projection * view * model * vec4(coordinate.xyz, 1.0);
```

De berekening van de locatie begint met het vermenigvuldigen van alle matrices. Daarna wordt op het einde een vector van vier componenten samengesteld vanuit het *x*, *y*, en *z*, coördinaat van de invoer, en een waarde van 1.0. Deze waarde is nodig voor de perspectief matrix, en zal geen invloed hebben op het uiteindelijke beeld. Tot slot kan de main functie, en string worden afgebakend. Hiermee is de Vertex Shader compleet.
In conclusie, de Vertex Shader krijgt data aangevoerd vanuit het videokaartgeheugen, deze data wordt geïnterpreteerd als een vector met vier componenten. Vervolgens wordt deze vector geprojecteerd naar een drie dimensionale vector, en nog een geprojecteerd naar de juiste locatie volgens het perspectief dat is ingesteld. Nu wordt de locatie doorgestuurd naar de Fragment Shader, en daar kan de kleur van het punt worden ingesteld.

Fragment Shader

Met de locatie van alle punten bepaald, kan er worden gekeken naar de kleur van de punten. Kleur in OpenGL is zeer uitgebreid, omdat het volgens het RGBA spectrum werkt. Dit spectrum kan richting de zestien miljoen verschillende kleuren genereren, en geeft dus veel mogelijkheden voor kleuren mappen van de fractal. Maar voor nu zal er worden gefocust op het inkleuren van de punten, en dit is relatief simpel gedaan. Net als de Vertex Shader begint de Fragment Shader als een string, en met een versie declaratie:

```
fragment_shader_string = '''
#version 330 core
```

Vervolgens moet de uitvoer worden gespecificeerd, dit zal in de vorm van een vector met vier componenten zijn. Elk component is representatief voor een waarde van RGBA, ofwel Rood, Groen, Blauw, en Alfa. Alfa kan worden gezien als de doorzichtigheid van een punt. De naam van deze vector zal worden aangegeven met 'FragColor':

```
out vec4 FragColor;
```

Vervolgens moet de *main()* functie weer worden beschreven. Hierin kan de kleurenmap worden gecodeerd. Voor nu zal alleen de kleur zwart worden gegeven aan elk van de punten:

```
void main(){
    FragColor = vec4(0, 0, 0, 1);
}
```

En daarmee is de meest simpel mogelijke Fragment Shader af. Nu hoeven de shaders alleen nog samen gekoppeld te worden tot een programma. En daarna kan de data worden ingevoerd voor de langverwachte fractal.

Shader Programma

Het Shader Programma is een relatief simpele stap in het hele proces, maar wel noodzakelijk. Anders begrijpen de videogeheugen kernen niet wat ze aanmoeten met de codering. Om te beginnen zal er weer een functie worden aangemaakt om deze stappen automatisch uit te voeren. De functie zal een lijst van alle shaders als invoer nemen:

```
def make_program(shader_list):
    '''Creates an OpenGL program with the given shaders. An OpenGL context
    must be active before calling this.'''
```

Zoals vermeld staat in de 'docstring', ofwel de tekst onder de functie, moet er eerst een OpenGL context actief zijn voordat deze functie kan worden aangeroepen.
Verder kan er nu een OpenGL programma worden gecreëerd, hier worden vervolgens de geschreven shaders aan gekoppeld:

```
    #Create the OpenGL program
    program = glCreateProgram()

    #Attach shaders to program
    for shader in shader_list:
        glAttachShader(program, shader)
```

Vervolgens moet het programma 'gelinkt' worden. Dit betekent dat het programma naar de videokaart wordt gestuurd, en daar wordt voorbereid voor uitvoer. Dit gebeurt door middel van de functie *glLinkProgram*. Het is ook handig om te kijken of het linken succesvol is verlopen, daarvoor kan een simpele test worden uitgevoerd die de status van het link proces terugkeert:

```
    #Link program
    glLinkProgram(program)

    #Checks if the linking succeeded
    linking_status = glGetProgramiv(program, GL_LINK_STATUS)
```

Linking_status kan nu twee waarden opleveren: Gl_FALSE, of GL_TRUE. Als de waarde onwaar is, dan is het linken mislukt, en dan moet de error worden opgehaald die OpenGL teruggeeft. Dit gebeurt door middel van *glGetProgramInfoLog()*:

```
    if linking_status == GL_FALSE:
        #Gets the error from OpenGL
        linking_error = glGetProgramInfoLog(program)
        print(f"Linking failure\nError: {linking_error}")
```

Uiteraard kan het linken ook succesvol verlopen, en een stukje feedback in de terminal hiervan is altijd fijn:

```
    else:
        print("Program linked successfully!")
```

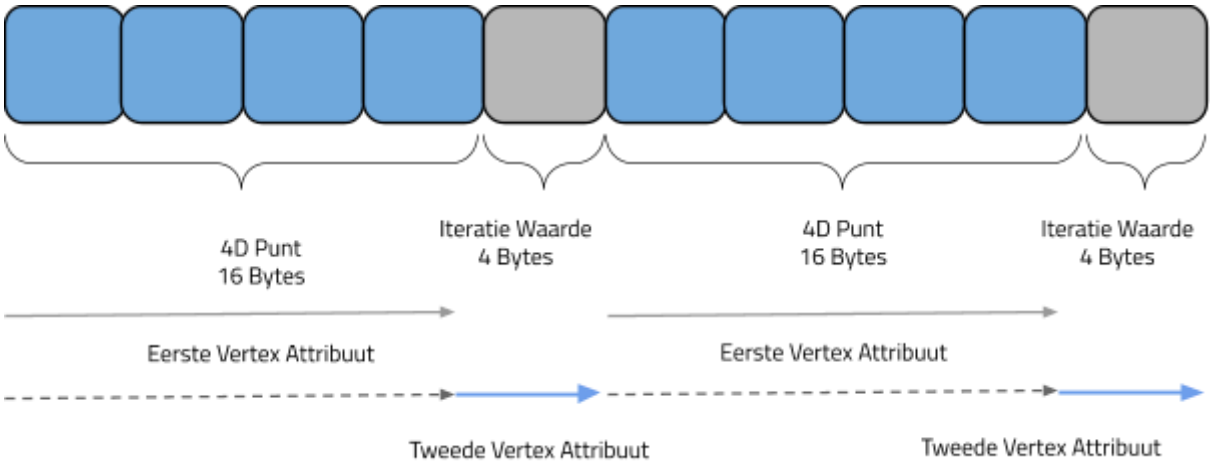
Tot slot kunnen de shaders weer worden losgekoppeld, en worden 'verwijderd'. Dit is om computatie kracht te besparen. Daarna kan het gelinkte programma worden teruggekeerd:

```
    #Detaches and deletes shaders once the program has been successfully
    linked
    for shader in shader_list:
        glDetachShader(program, shader)
        glDeleteShader(shader)
    return program
```

Nu is het Shader Programma klaar voor gebruik, nu hoeft alleen nog de data worden opgezet voor OpenGL om te begrijpen, en dan kan er gerenderd worden.

Opzet van OpenGL data

OpenGL werkt door middel van 'Vertex Buffer Objects' (VBO's), dit zijn buffers die alle data bevatten. Deze buffers werken samen met 'Vertex Attribute Objects' (VAO's), dit zijn instructies over hoe de data uit de VBO moet worden opgevat. De data die origineel is gegenereerd bestaat uit vierdimensionale punten, samen met iteratie waarden. De punten zijn opgeslagen als decimale getallen, deze decimale getallen zijn elk vier bytes, dus totaal is een punt zestien bytes aan data. De iteratie waarden staan ook opgeslagen als decimale getallen, en nemen in totaal dus vier bytes op. Het doel is om deze data apart op te vatten, zodat uiteindelijk het aantal iteraties kan zorgen voor een kleurenmap, of andere functies. Er moet dus onderscheid worden gemaakt tussen bepaalde locaties van het geheugen.
Dit gebeurt door middel van zogenaamde 'Vertex Attributen', en deze attributen worden opgeslagen in de VAO's, zodat OpenGL ze gemakkelijk kan lezen.
De data die in de VBO wordt gestopt komt uit een array, en arrays zijn continue in geheugen. Dat betekent alle waardes naast elkaar liggen in het geheugen. Dus kan dit concept in zijn geheel worden opgevat als de volgende afbeelding:



Nu moet dit alles nog aan OpenGL worden verteld, daarvoor wordt er weer gebruik gemaakt van een nieuwe functie, die de data als invoer neemt:

```
def data_setup(data):
    '''Initializes and sets up all data needed for the OpenGL rendering'''
```

Vervolgens kan er een VBO en VAO worden gegenereerd, en deze worden daarna direct gekoppeld aan de juiste variabelen binnen OpenGL:

```
    #Generation of vertex array. Which holds attributes. Then binds it
    VAO = 1
    glGenVertexArrays(1, VAO)
    glBindVertexArray(VAO)
    #Generate a buffer with an unique ID and binds it to the right buffer
    VBO = 1
    glGenBuffers(1, VBO)
    glBindBuffer(GL_ARRAY_BUFFER, VBO)
```

Daarna kan de data worden ingevoerd in de gemaakte VBO, dit gebeurt door middel van *glBufferData()*. Deze functie neemt vier argumenten aan:

- Het soort buffer waar de data in moet worden opgeslagen, in dit geval een 'GL_ARRAY_BUFFER'.
- De grootte van de data in bytes. Dit kan worden achterhaald door de *.itemsize* en *.size* methodes van Numpy.
- De data die in de buffer moet worden gedaan.
- Wat voor soort data het is. Als de data bijvoorbeeld niet wordt veranderd tijdens het renderen dan wordt er gebruik gemaakt van 'GL_STATIC_DRAW'. Als de data wel veel verandert kan er gebruik worden gemaakt van 'GL_DYNAMIC_DRAW'. Dit variabel is dus contextueel aan de data.

De uiteindelijke functie komt er dan zo uit te zien:

```
    #Copy the data into the buffer, with the according byte size.
    glBufferData(GL_ARRAY_BUFFER, vertex_data.size * vertex_data.itemsize,
    vertex_data, GL_STATIC_DRAW)
```

Nu kunnen de vertex attributen worden gedefinieerd, en dit gebeurt door middel van *glVertexAttribPointer()*. Deze functie neemt zes argumenten aan:

- Het indexgetal van het vertex attribuut. Dit index getal wordt dan ook gebruikt in de shader voor de '*layout call*'.
- Het aantal punten waaruit de data bestaat, dus in het geval van het vierdimensionale punt is dat vier. En voor de iteratie waarde is het één.
- Het type data in OpenGL vorm, dus 'GL_FLOAT'.
- Een optie of de data genormaliseerd moet worden. Dit hoeft niet, want het gebeurt al door middel van de projectiematrix.
- De grootte van het eerste attribuut tot de volgende in bytes. Dus niet het tweede attribuut, maar hetzelfde attribuut maar dan later in het geheugen.
- De locatie waar moet worden begonnen met het lezen van het geheugen.

Zoals eerder achterhaald moet het coördinaten vertex attribuut dus bestaan uit vier decimalen, en het iteratie attribuut komt daarachter met een decimaal. Dus de afstand voordat het volgende coördinaten attribuut wordt bereikt is gelijk aan vijf keer vier bytes, dus twintig.
Het coördinaten attribuut zal worden gelabeld met een nul, en het iteratie attribuut met één.
Het iteratie attribuut moet vanaf zestien bytes pas worden gelezen, dus de locatie is zestien van het tweede attribuut. Dit resulteert in:

```
    #Creates a vertex attribute with index 0, which is also specified in the
    shader with '(layout = 0)'
    #Coordinate attribute
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE,
    5 * vertex_data.itemsize, None) #Works

    #Iteration attribute (Attribute offset needs to be in c-class)
    glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE,
    5 * (vertex_data.itemsize), c_void_p(4 * vertex_data.itemsize))
```

Nu hoeven de attributen alleen nog geactiveerd te worden, en gekoppeld te worden aan de juiste buffer. Daarna kan de VBO, en VAO worden teruggestuurd om te gebruiken in de Render functie:

```
    #Enables the vertexAttribArray with index 0 & 1
    glEnableVertexAttribArray(0)
    glEnableVertexAttribArray(1)

    glBindBuffer(GL_ARRAY_BUFFER, 0)
    glBindVertexArray(0)

    glBindBuffer(GL_ARRAY_BUFFER, 1)
    glBindVertexArray(1)

    return VAO, VBO
```

En hiermee is de data klaar voor gebruik. Nu hoeft OpenGL alleen nog de juiste render functies uit te voeren, en dan is er beeld.

De 'Render' functie

Hoogstwaarschijnlijk is het te voorspellen waarmee begonnen zal worden, indien dat niet zo is: Een nieuwe functie natuurlijk!

```
def render(window, display, VAO, VBO, shader_program):
    '''Functions to render the dataset'''
```

De render functie neemt als invoer de window die is gecreëerd door glfw, de breedte en hoogte van het scherm, de VAO en VBO, en het shader programma. Ofwel alle delen die tot nu toe zijn gegenereerd.

Vervolgens kunnen een aantal globale OpenGL opties worden geregeld, zoals het testen van diepte, en de grootte van punten:

```
#OpenGL global setup
glEnable(GL_DEPTH_TEST)
glEnable(GL_PROGRAM_POINT_SIZE)
```

Daarna moeten de matrices nog worden opgesteld die zijn gebruikt in de Vertex Shader. De eerste matrix is de model matrix, en deze kan, zolang er geen translaties of rotaties nodig zijn, gewoon een identiteitsmatrix blijven:
De tweede matrix is de perspectief matrix, deze matrix zal bestaan uit een translerende transformatie. Die de camera vijf eenheden op de z-as verplaatst:
Tot slot moet de projectiematrix worden gedefinieerd, hiervoor heeft de glm module gelukkig een ingebouwde functie. Namelijk *glm.perspective()*, in deze functie moeten vier argumenten worden toegevoerd:

- Het gezichtsveld in radialen, of in graden die worden omgezet naar radialen door middel van *glm.radians()*. Een gezichtsveld van 45 graden is voldoende voor nu.
- Het aspect ratio van het scherm, hiervoor kan de *display* lijst worden gebruikt die wordt ingevoerd.
- Het minimale en maximale knipvlak, hiervoor zijn de waardes 0.1 en 100 prima.

Dit alles resulteert in:

```
#Setup of matrices for true 3D
model_matrix = glm.mat4(1)

view_matrix = glm.mat4(1)
view_matrix = glm.translate(view_matrix, glm.vec3(0, 0, -5))

projection_matrix = glm.mat4(1)
projection_matrix = glm.perspective(glm.radians(45.0),
display[0]/display[1], 0.1, 100)
```

Met de matrices in plaats kan er een lus worden gestart waarin de render functies komen. De lus zal blijven draaien totdat de gebruiker het gecreëerde scherm wegklikt, daarna stopt het programma:

```
while not glfw.window_should_close(window):
  ...
```

Om nu aan het concrete renderen te beginnen moet eerst het scherm worden gewist, zodat nieuwe renders duidelijk naar voren. Hiervoor dient *glClearColor()*, deze functie wist het gehele scherm met de kleur die wordt ingesteld in de functie, deze kleur is in RGBA vorm. Daarna moet ook de 'Color buffer' worden geleegd, en de 'Depth buffer', zodat er geen artefacten ontstaan, en zodat diepte juist wordt gerenderd:

```
    #Render Commands Here
    glClearColor(1,1,1,1)       #White Background
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

Daarna moet OpenGL worden verteld welk Shader Programma gebruikt moet worden, dit gaat via *glUseProgram()*. Verder moet ook de locatie in het geheugen van de uniformen worden bepaald, met deze locatie kan daarna het uniform naar die locatie worden verplaatst, en worden gebruikt:

```
        #Lets OpenGL use the program made in 'make_program'
        glUseProgram(shader_program)

        #OpenGL Uniform pass
        model_location = glGetUniformLocation(shader_program, "model")
        view_location = glGetUniformLocation(shader_program, "view")
        projection_location = glGetUniformLocation(shader_program,
        "projection")
```

Om de matrices naar de locatie van het uniform te sturen moet er gebruik worden gemaakt van *glUniformMatrix4fv()*. Deze functie werkt met vier argumenten:

- De locatie van het uniform, die eerder achterhaald is.
- Het aantal variabelen wat wordt toegestuurd, dus een.
- Een mogelijkheid om de matrices te transponeren, wat niet nodig is.
- Een 'pointer', ofwel verwijzing naar geheugen, van het variabel dat het uniform representeert. Python gebruikt geen pointers, daarom wordt er gebruik gemaakt van de glm functie *value_ptr()*.

```
        #Send matrices to uniforms
        glUniformMatrix4fv(model_location, 1, GL_FALSE,
        glm.value_ptr(model_matrix))
        glUniformMatrix4fv(view_location, 1, GL_FALSE,
        glm.value_ptr(view_matrix))
        glUniformMatrix4fv(projection_location, 1, GL_FALSE,
        glm.value_ptr(projection_matrix))
```

Daarna moet OpenGL worden verteld welke VAO er wordt gebruikt. Daarna kan de functie *glDrawArrays*() worden gebruikt om de gecreëerde arrays met data te gaan tekenen. Deze functie werkt door middel van drie argumenten:

- Het soort primitieven om te tekenen, in dit geval is dat 'GL_POINTS'.
- Het index getal van de VBO om te gebruiken als data, dus nul.
- Het maximum aantal punten dat mag worden getekend, ofwel een heleboel.

```
        #Render
        glBindVertexArray(VAO)
        glDrawArrays(GL_POINTS, 0, 10000000)
```

Tot slot hoeft alleen nog de 'buffer te worden verwisseld'. Dat impliceert dat het beeld een update krijgt. Dit kan worden gedaan door middel van *glfw.swap_buffers()*:

```
        glfw.swap_buffers(window)
```

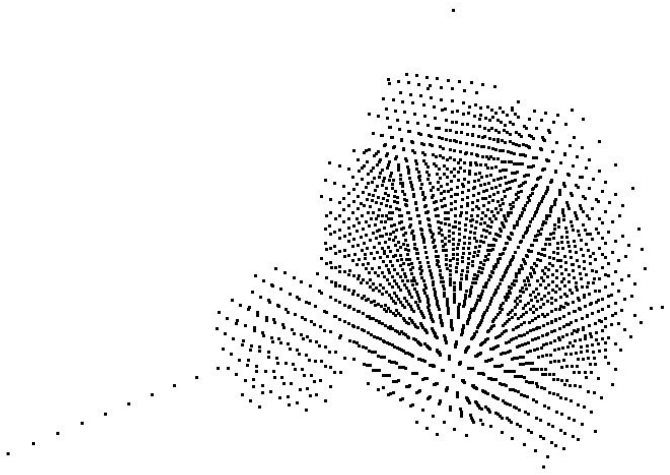
Daarmee is de render functie ook af, nu moet alles nog netjes in *main()* worden gezet, en dan kan er naar het beeld gekeken worden!

De 'Main' functie

Met alle functies uitgewerkt en geprogrammeerd kan alles worden aangeroepen. Dit is een simpel process van de functies op volgorde zetten binnen de *main()* functie, met de juiste variabelen:

```
def main():
    #Variable setup
    display = [1200, 1000]

    #Data generation
    combined_dataset = dataset_generator(fineness = 50, filter_mode = True,
    save_mode = True)

    #Window, shader, and program setup
    window, display = make_window(display[0], display[1])
    shader_program = make_program([vertex_shader_string,
    fragment_shader_string])

    #OpenGL VBO/VAO data setup
    VAO, VBO = data_setup(combined_dataset)

    #Render call
    render(window, display, VAO, VBO, shader_program)
```

Het hele process begint bij het aanmaken van een grote dataset, van vijftig bij vijftig bij vijftig bij vijftig, deze dataset wordt dan gefilterd en opgeslagen voor later gebruik.
Vervolgens wordt het scherm aangemaakt, samen met het shader programma. Daarna wordt de VBO en VAO opgesteld met de dataset gegenereerde dataset. En tot slot wordt de dataset gerenderd:

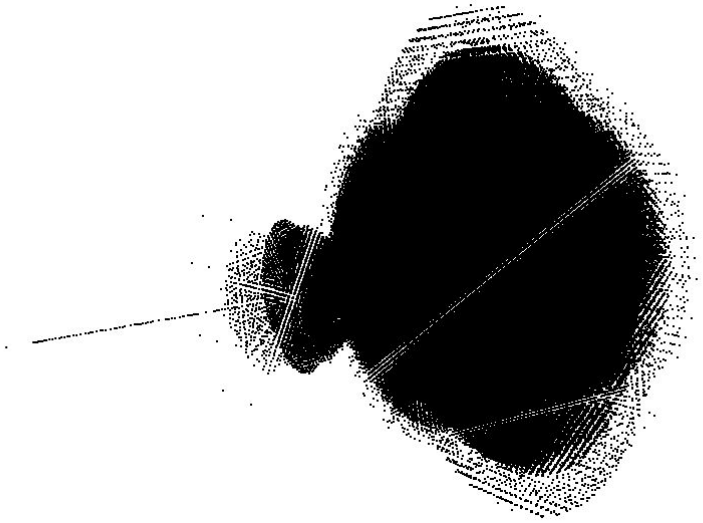


Conclusie

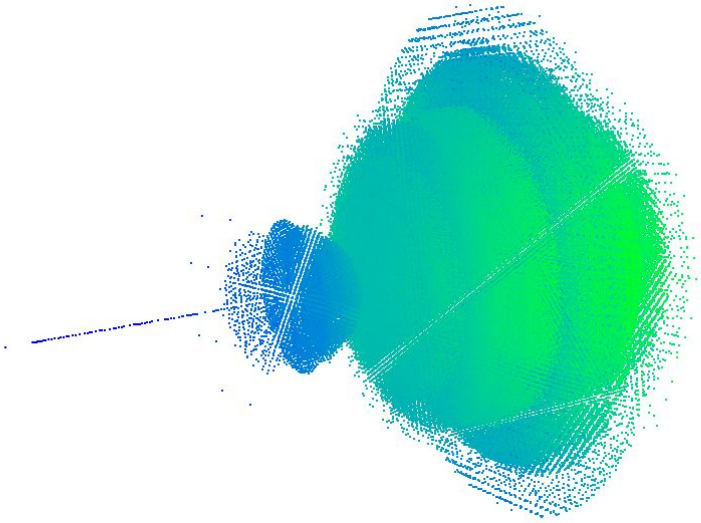
In conclusie is er dus een heleboel nodig om een fractal uit de vierde dimensie naar de derde dimensie te halen, en te renderen op een tweedimensionaal scherm. Maar het is niet onmogelijk! Hieronder zijn nog een aantal beelden te zien van datasets met verschillende projecties, kleuren, en meer. Elke afbeelding zal kort worden toegelicht, en daarna zal het onderzoek worden geconcludeerd.

Hogere resolutie

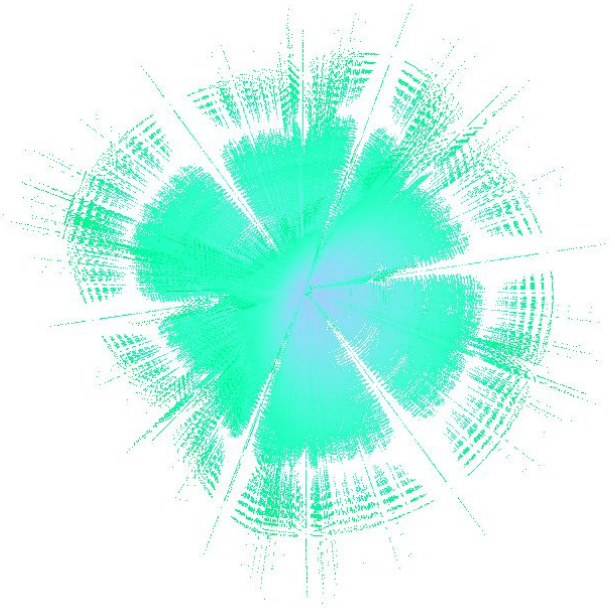
De dataset die de afbeelding op de vorige pagina genereerde was slecht op een resolutie van 50^4. Dit kan uiteraard beter, de onderstaande afbeelding is op een resolutie van 500^4 gegenereerd.



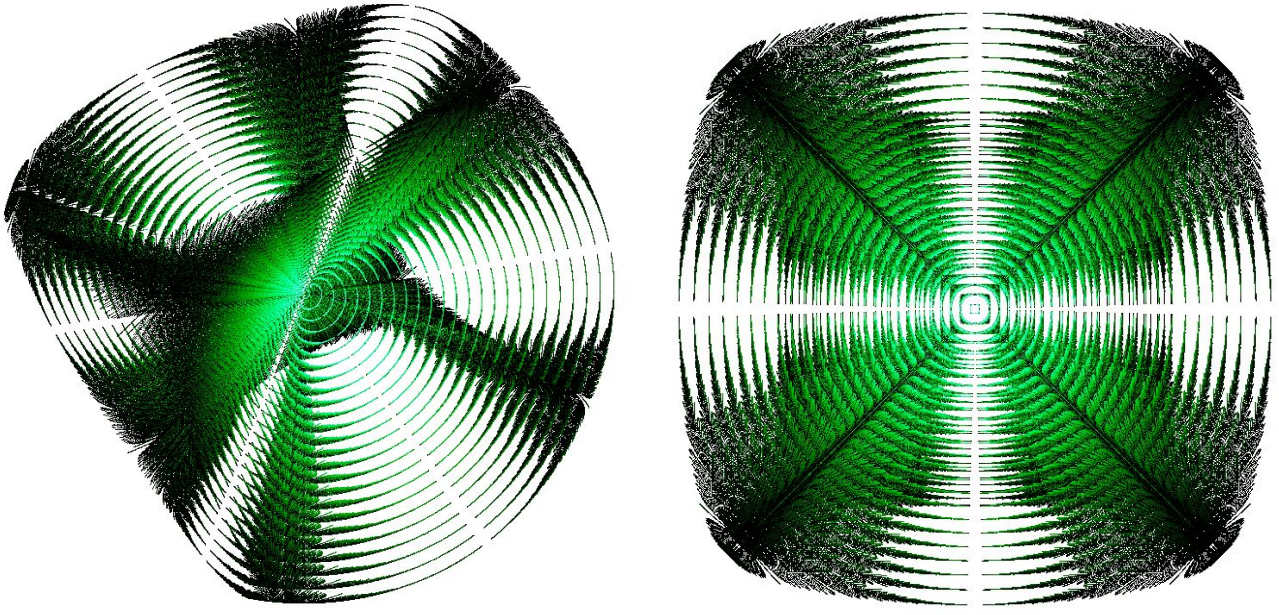
Helaas is er weinig van te zien doordat alle zwarte punten in elkaar overlopen. Een kleine aanpassing in de Fragment Shader kan hiermee helpen, deze stap zal daarentegen niet worden doorlopen. Alle codering is te vinden in de bijlage. Hiernaast wordt elk punt gekleurd ten opzichte van de afstand van de oorsprong.



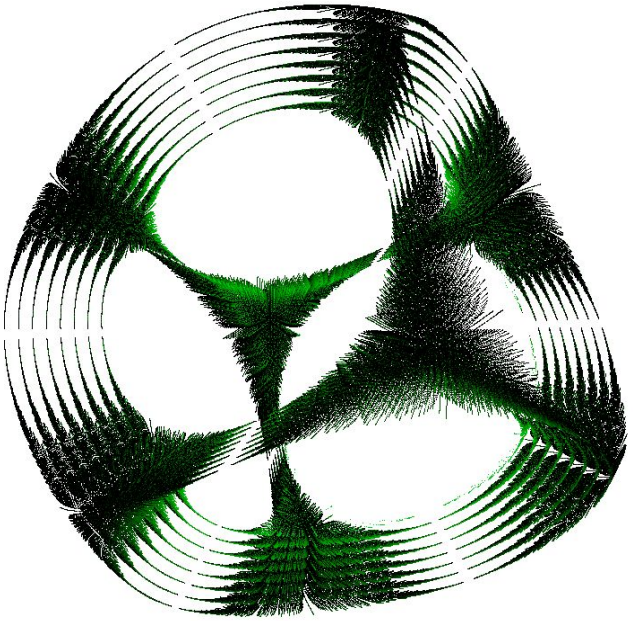
Uiteraard moet ook nog de geometrische projectie worden bekeken. Hiervoor wordt de dataset door de geometrische projectie functie gehaald, en de dataset die daar uit komt wordt gerenderd. Het is niet de meest accurate projectie, maar het levert zeker wel een interessant resultaat op.



Om het volledige effect van de geometrische projectie te zien is er een dataset in de vorm van een vierdimensionale kubus de functie ingevoerd. Het resultaat hiervan is zeker niet lelijk:



Gebruikte Vertex Shader: geometric_projection_vertex_shader_string
Dataset: dataset_generator(fineness = 50)

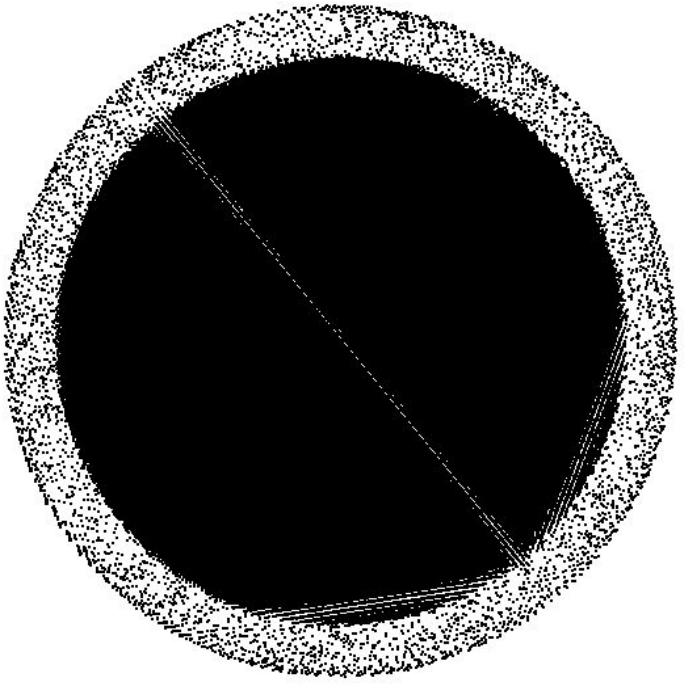


De geometrische projectie met de helft van alle punten gerenderd

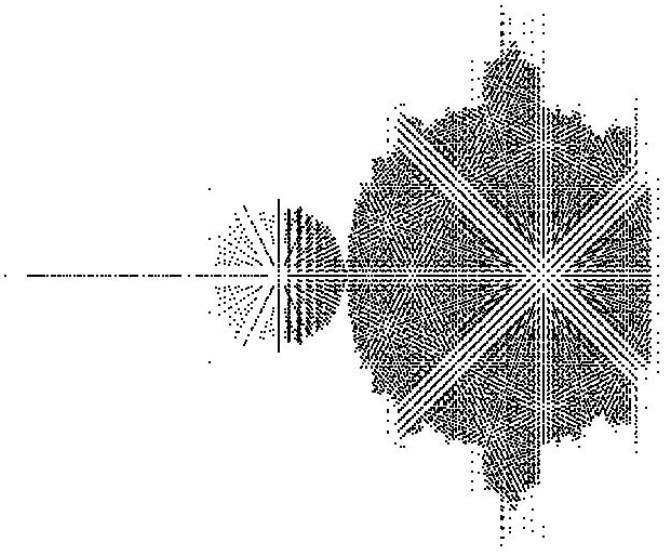
Tot slot zullen er nog een aantal projecties worden weergegeven die verschillende matrices gebruiken als projectie. Dit process is gelijk aan het verwijderen van een van de dimensies van de vierdimensionale punten:

1) Gek genoeg levert de projectie van de matrix hiernaast niks anders op dan een bol. Dus wanneer het complexe deel van de vierdimensionale Mandelbrot Set wordt genomen levert dit een driedimensionale bol op.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

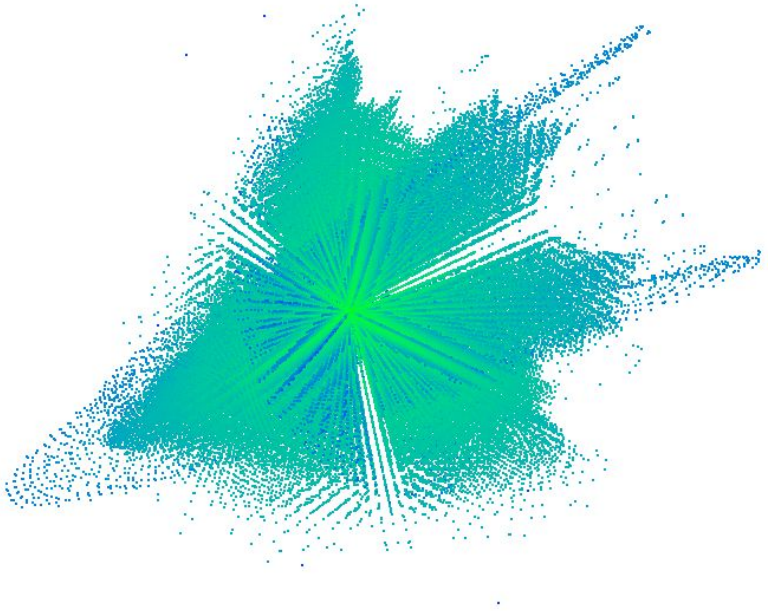


2) Als de projectiematrix twee complexe dimensies weglaat, komt er een bekend plaatje naar voren. De Mandelbrot Set! Wel in lage resolutie, maar alsnog een interessant resultaat.



3) Tot slot is er in de Vertex Shader het volgende proces uitgevoerd:
   a) $\{a, b, c, d\} \in Q$
   b) X = a*b, Y = b*c, Z = c*a
   c) d => Bepalend voor de kleur.

   En dat levert dit resultaat op:



~En zo zijn er nog vele andere mogelijkheden. Zie de galerij voor meer plaatjes.

Optimalisatie van het programma

Tot slot zijn er nog een aantal onbeantwoorde vragen over OpenGL, en is er nog steeds geen concreet antwoord op de optimalisatie deelvraag. Er zal eerst worden gekeken naar de werking van OpenGL, en hoe dit samenhangt met de optimalisatie van het programma. Vervolgens zullen een aantal methoden worden besproken om het programma nog verder te optimaliseren, een groot aantal van deze methoden komen van Michael McKerns's Scipy Tutorial.

OpenGL buffers

OpenGL maakt gebruik van zogenaamde "buffers". Deze buffers zijn blokken van het geheugen waar de verwerkte data wordt in opgeslagen tijdens het process van renderen. De reden voor deze tussenstap is snelheid, en om dit beter te begrijpen moet eerst nog een ander concept worden geïntroduceerd. Namelijk "FPS", oftewel frames per seconden. Dit is het aantal beelden per seconde dat de computer naar het beeldscherm stuurt. Vervolgens is er nog een "refresh rate" van het beeldscherm, dat impliceert hoeveel beelden het scherm zelf kan laten zien. Betere beeldschermen hebben daarom ook een hogere "refresh-rate". En de reden waarom OpenGL de data opslaat in buffers is zodat de uiteindelijke frames kunnen worden opgeslagen. Door middel van deze aanpak kan het renderen dus voor gaan lopen op wat het beeld laat zien. Als er bijvoorbeeld dertig frames per seconde worden weergegeven op het scherm, en OpenGL zestig frames per seconde rendert. Dan zal OpenGL per seconde dus dertig frames voorlopen. En dit is een algehele optimalisatie voor het programma.

Speciale functies van numpy

Numpy is al enorm goed in het versnellen van data-intensieve calculaties, maar zoals eerder besproken in "*Het generatieve programma*" zijn er speciale functies, zoals *np.square()* die sneller zijn dan normale Python functies. Dit komt omdat deze speciale functies, die overigens *Ufuncs* worden genoemd, stukjes C codering zijn, die zijn gecompileerd. Dit maakt de functie sneller omdat C geen geïnterpreteerde taal is zoals Python. Er kan ook wel worden gezegd dat C dichter tot de computer zelf staat. Hierom is het dus verstandig om zoveel mogelijk ingebouwde functies van Numpy te gebruiken.

Management van data

Uiteraard is het opslaan van berekende data een zeer goede manier om het gebruik van het programma te optimaliseren, hierdoor hoeft niet er niet per render een hele nieuwe dataset te worden gegenereerd. Ook is het filteren van de data een grote optimalisatie geweest, in sommige gevallen is de uiteindelijke dataset slechts 1% van de berekende set.

Conclusie

Dit hoofdstuk markeert het einde van dit onderzoek, *voor nu.* Tot slot zal elke onderzoeksvraag nog eens worden beantwoord in complete context. Het antwoord op de hoofdvraag zal ook worden vergeleken met de originele hypothese.

Deelvragen

Wat is een fractal, en welke eigenschappen heeft dit wiskundige object?

Een fractal is een wiskundig object dat vormen vertoond die soortgelijk zijn aan het geheel naarmate de schaal veranderd. Fractals hebben ook een intrinsiek patroon dat niet zomaar kan worden beschreven in normale geometrische termen. Een fractal kan ook worden gedefinieerd vanuit een puur wiskundige definitie, die stelt dat de Hausdorff dimensie van de fractal groter moet zijn dan de topologische dimensie, ofwel:

$$F \subseteq \mathbb{R}^n \mid n \in \mathbb{Z},\ dim_H(F) > dim(F)$$

Verder zijn fractals relatief simpel te genereren, en kunnen voortkomen uit een systeem van simpele regels, of functies. Tot slot heeft een fractal een zekere fractale dimensie, ofwel een getal dat impliceert hoe complex het detail van de fractal is naarmate de schaal veranderd.

Hoe wordt een fractal gegenereerd, en hoe kan dit generatie process worden beïnvloed?

Een fractal kan op enorm veel verschillende manieren worden gegenereerd, en komen in allerlei contexten voor, van het modelleren van konijnenpopulaties, tot het itereren van complexe getallen. De drie voornaamste manieren van fractal generatie zijn:

- Iteratieve functie systemen: Hier wordt door een iteratie van een bepaalde regel of manipulatie een fractale vorm gegenereerd, zoals de Sierpinski driehoek.
- Vreemde aantrekkers: Binnen dynamische systemen kunnen fractals voorkomen door de juiste weergave van waardes van het dynamisch systeem, zoals de Feigenbaum Aantrekker.
- Ontsnappingstijd fractals: De methode waar een verzameling van coördinaten wordt geïtereerd, en aan bepaalde condities moet voldoen om deel te zijn van de uiteindelijke fractal. Zoals de Julia Set.

Het is ook mogelijk om door middel van speciale sets van getallen, zoals de quaternions, fractal te genereren in hogere dimensies. Het process hiervan gebeurt via dezelfde methode als het genereren van de Mandelbrot Set, en resulteert in een vierdimensionale Mandelbrot Set

Hoe kan het gebruik van hogere dimensionale datasets worden geoptimaliseerd?

Optimaliseren kan op vele verschillende manieren, en elke methode heeft een ander resultaat. Een aantal manieren van het optimaliseren van Python's gebruik van multidimensionale datasets zijn:

- Gebruik van speciale modules die de "trage" delen van Python vervangen met snelle gecompileerde C codering. Dit gebeurt bijvoorbeeld binnen Numpy. En resulteert in een toename in uitvoeringssnelheid van het programma.
- Het opslaan van verwerkte data scheelt veel tijd op lange termijn. Omdat de data niet elke keer opnieuw gegenereerd hoeft te worden.
- Data kan worden gefilterd voor nuttige onderdelen, de rest kan weg worden gegooid om geheugen en computatie tijd te besparen.
- Het gebruik van gespecialiseerde software, ofwel CUDA en OpenGL, kan Python enorm versnellen. Dit komt voornamelijk doordat beide CUDA en OpenGL gebruik maken van parallelle computatie op de GPU. In tegenstelling tot lineaire computatie op de CPU. Dit resulteert in de mogelijkheid om veel data tegelijkertijd te renderen, zonder grote render-tijden.

In conclusie, optimalisatie is voornamelijk een kwestie van kennis over de computer en software. En kan op vele manieren worden uitgevoerd.

Hoofdvraag

Hoe is een multidimensionale fractal te visualiseren in een lagere dimensie?

Multidimensionale objecten zijn te visualiseren in lagere dimensies door middel van lineaire algebra. Dit gaat door middel van een projectiematrix die ervoor zorgt dat een *n-dimensionale* vector, wordt gereduceerd tot een *(n-1)-dimensionale* vector. De projectiematrix kan orthogonaal zijn, dat impliceert dat het beeld dat uit de projectie komt alle parallelle eigenschappen van het origineel heeft behouden. Een orthogonale projectie moet aan een aantal eisen voldoen, namelijk:

- $P^2 = P$
- $P = P^T$
- De vectorruimte V waarvan P projecteert moet compleet zijn.

De projectie kan ook orthogonaal zijn, dat impliceert dat de parallelle eigenschappen van het origineel verloren zijn gegaan tijdens de projectie.
Verder is het belangrijk om op te merken dat er vele soorten projectie matrices zijn, en dat verschillende matrices verschillende projectieve resultaten hebben.
Aan het begin van het onderzoek is de volgende hypothese opgesteld:
"*Het is mogelijk om de multidimensionale objecten naar een lagere zichtbare dimensie te brengen door middel van lineaire algebra. Dit wordt gedaan door middel van een matrix transformatie. De eigenschappen van de fractal die bewaard blijven in de lagere dimensie, is afhankelijk van de soort matrix die wordt gebruikt.*"

Deze hypothese is dus bewezen, het is alleen niet zo netjes verwoord. Maar dat laat wel goed de voortgang zien van het onderzoek.
Er moet wel worden opgemerkt dat het ook mogelijk is zonder lineaire algebra hogere dimensies te projecteren naar lagere. Alhoewel de resultaten van dit soort projecties nog moeten worden onderzocht. Dit zou een potentieel vervolgonderzoek kunnen zijn.
Hiermee is het onderzoek afgerond, en zijn alle onderzoeksvragen beantwoord. Tot slot zal er nog een discussie worden gehouden over hoe dit onderzoek beter had kunnen gaan.

Discussie

Dit onderzoek had op een aantal punten zeker wat limitaties. En hoewel dit onderzoek uitgebreid is, heeft het nog niet eens een fractie van beide fractals, en het concept van multidimensionaliteit behandeld. Ook zijn beide onderwerpen op zichzelf vrij diep uitgewerkt, maar de combinatie van de twee is minder naar voren gekomen.
Verder was mijn programmeerervaring eigenlijk te weinig om dit onderzoek goed te kunnen houden. En dit heeft meer een beperking gehad op het benut van de tijd, dan een beperking op het onderzoek. Ondanks dat is mijn programmeer ervaring wel met enorme mate toegenomen.

De verkregen antwoorden op de onderzoeksvragen kunnen worden geïnterpreteerd als een fundatie voor een toekomstig, nog dieper onderzoek. Dit gehele onderzoek zelfs als vooronderzoek kunnen worden gebruikt voor een uitgebreider onderzoek over de multidimensionaliteit van fractals, en alle bijbehorende takken van wiskunde die hiermee te maken hebben. Dat niet alleen, het programma wat is geschreven heeft veel ruimte voor uitbreidingen en optimalisatie. Een toekomstig vervolgonderzoek zou dus bestaan uit alles dat hier is besproken, en hierop uitbreiden. Voornamelijk op het concept van dimensies, en de eigenschappen die hiermee te maken hebben.

Dankwoord
Tot slot wil ik, Michael Schuchter, mijn dank uiten aan een groep mensen. Die allemaal hebben meegeholpen in het process van dit onderzoek.

Iris Loeb
Mijn eerste en grootste dank gaat uit naar Mevrouw Loeb. Zij heeft veel gekke en absurde wiskunde vragen beantwoord tijdens lessen, en daarbuiten. Ook heeft zij de begin versies van dit onderzoek met concrete feedback voorzien. Dat zeker heeft geholpen met het finaliseren van het verslag.

Tristan van Grieken
Tristan is een oude klasgenoot, en zeer goede vriend van mij. Hij heeft mij vaak geholpen met programmeer problemen, en heeft nuttige kritiek geleverd op de programmeer delen van dit onderzoek. Verder is hij altijd zeer enthousiast geweest over de gegenereerde afbeeldingen, en het uiteindelijke verslag.

Tesse Brandt
Mijn vriendin, die me vaak heeft gemotiveerd om verder te werken wanneer het tegen zat. Zij heeft ook het verslag voorzien van feedback en kritiek, die later is verwerkt tot het uiteindelijke resultaat.

Ouders
Voor het steunen van mij tijdens mijn onderzoek, bijvoorbeeld door koffie, of tosti's voor mij klaar te maken. *:)*

Een laatste woord
Dit onderzoek was zeer interessant om te houden, en ik heb er zeker van genoten. De resterende documentatie, logboek, en galerij is te vinden in de bijlage van dit document. En op de laatste pagina's.

Abstract
This research focuses on a variety of subjects, ranging from fractal generation to projections of higher dimensions to lower ones. But this paper mainly describes how to visualize a higher dimensional fractal to a lower dimension through the means of linear algebra. The research conducted to come to this conclusion was both theoretical, and practical. The theoretical research in the context of literary research, and the practical research focused on using Python to verify the theoretical research. Furthermore Python and OpenGL are used to visualize a variety of fractals, which are pretty to look at, and form a good source for mathematical research.

Literatuurlijst

Experts:

- Tristan van Grieken, voor een variatie aan Python gerelateerde vragen
- Henk Stitselaar, voor een aantal vragen over Numpy
- Iris Loeb, voor een variatie aan vragen
- Remi Blokker, voor een aantal vragen over coderen

Literatuur

- 3Blue1Brown "Visualizing quaternions with stereographic projection" (2018)
    - https://www.youtube.com/watch?v=d4EgbgTm0Bg
- Alfeld, P. "The Mandelbrot Set."
    - http://www.math.utah.edu/~alfeld/math/mandelbrot/mandelbrot.html
- Benoit Mandelbrot, "The Fractal Geometry of Nature" (1982)
- Carlos Madrid, "Chaostheorie en klimaatverandering" (2013)
- Eaglieview.com voor verscheidene afbeeldingen.
- Edyta Patrzalek, "General introduction to Fractal Geometry" (2012)
    - http://www.fractal.org/Bewustzijns-Besturings-Model/Fractals-Useful-Beauty.htm
- Encyclopedie van wiskunde, "Division algebras" (2012)
    - https://www.encyclopediaofmath.org/index.php/Division_algebra
- Harold Serrana, "How does a rendering engine work" (2016)
    - https://www.haroldserrano.com/blog/how-to-develop-a-rendering-engine-an-overview
- Holly Krieger, Numberphile "The Mandelbrot Set"
    - https://www.youtube.com/watch?v=NGMRB4O922I
- J.H. Dijkhuis, CJ Admiraal, JA Verbeek, G de Jong, HJ Houwing, F ten Klooster, SKA de Waal, J van Braak, JHM Liesting Maas, M Wieringa, MLM van Maarseveen, RD Hiele, JE Romkes, M Haneveld, S Voets, I Cornelisse. "Getal & Ruimte VWO D deel 3" (2016)
- Jake van der Plas, Pycon (2015). "Losing your loops: Fast numerical computing with Numpy"
    - https://www.youtube.com/watch?v=EEUXKG97YRw
- Jimi Sol, "How the Julia Sets work, and why it's amazing" (2017)
    - https://www.youtube.com/watch?v=mg4bp7G0D3s
- John Kenneth Falconer, "Fractal Geometry: Mathematical foundations and applications" (1990)
- Juan Pablo Xandri, "Notes on Metric Spaces"
    - https://scholar.princeton.edu/sites/default/files/jxandri/files/notes_on_metric_spaces_0.pdf

- Matplotlib documentatie, onderhouden door John Hunter, Darren dale, Eric Firing, Michael Droettboom, en de Matplotlib ontwikkelaars. (2019)
    - https://matplotlib.org/contents.html
- Michael Boyle, documentatie van Numpy-Quaternions (2019)
    - https://pypi.org/project/numpy-quaternion/
    - https://numpy-quaternion.readthedocs.io/en/latest/
- Michael McKerns, "Modern Optimization Methods in Python | SciPy 2017 Tutorial"
    - https://www.youtube.com/watch?v=geFER2oVvvU
- Nicolas P. Rougier "Python & OpenGL for scientific visualizations"
    - https://www.labri.fr/perso/nrougier/python-opengl/
- Numpy Documentatie, die grotendeels is gebaseerd op het boek: "Guide to Numpy" van Travis E. Oliphant. (2019)
    - https://docs.scipy.org/doc/numpy/
- Nvidia. "GPU Accelerated Computing with Python" (2019)
    - https://developer.nvidia.com/how-to-cuda-python
- OpenGL API documentatie
    - http://docs.gl/
- Pillow documentatie, onderhouden door Alex Clark en verscheidene contributeurs. (2019)
    - https://pillow.readthedocs.io/en/stable/
- Robert L Devaney (1995) "Fractal Dimension"
    - http://math.bu.edu/DYSYS/chaos-game/node6.html
- SourceForge, PyOpenGL documentatie
    - http://pyopengl.sourceforge.net/documentation/index.html
- Steve Mould, Numperphile "Measuring Coastline" (2014)
    - https://www.youtube.com/watch?v=7dcDuVyzb8Y
- The Cherno, "Using Modern OpenGL in C++"-series. (2017)
    - https://www.youtube.com/watch?v=H2E3yO0J7TM
- Veritasium, "This equation will change how you see the world"
    - https://www.youtube.com/watch?v=ovJcsL7vyrk
- Wikipedia is als algemene kennis bron gebruikt, en vrijwel niet voor directe kennis. Wikipedia is een open platform met frequente updates, dus zijn vrijwel alle bronnen van Wikipedia in de era van (2015-2019)
    - https://en.wikipedia.org/wiki/Fractal
    - https://en.wikipedia.org/wiki/Mandelbrot_set
    - https://en.wikipedia.org/wiki/Fractal_dimension
    - https://en.wikipedia.org/wiki/Dimension
    - https://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension
    - https://en.wikipedia.org/wiki/Fractal
    - https://en.wikipedia.org/wiki/Logistic_map
    - https://en.wikipedia.org/wiki/Logistic_function#Logistic_differential_equation
    - https://en.wikipedia.org/wiki/Bifurcation_diagram
    - https://en.wikipedia.org/wiki/Feigenbaum_constants
    - https://en.wikipedia.org/wiki/Mandelbulb
    - https://en.wikipedia.org/wiki/Frobenius_theorem_(real_division_algebras)
    - https://en.wikipedia.org/wiki/Abstract_algebra
    - https://en.wikipedia.org/wiki/Quaternion

    - https://en.wikipedia.org/wiki/Projection_(linear_algebra)
    - https://en.wikipedia.org/wiki/Linear_map
    - https://en.wikipedia.org/wiki/Vector_space
    - https://en.wikipedia.org/wiki/3D_projection
- Wolfram Mathworld, “Division Algebra”(2018)
    - http://mathworld.wolfram.com/DivisionAlgebra.html

Plan van Aanpak

Naam:	Michael Schuchter
Klas:	5V2 -> 6V3
Profiel:	N&T
Vak:	Wiskunde
Begeleider:	I. Loeb
Thema:	Hogere dimensionale fractalen, en de visualisatie daarvan
Type onderzoek:	Ontwerp / bronnenonderzoek

Motivatie:
Wiskunde is mijn favoriete vak, fractalen zijn erg mooi en leuk om naar te kijken, en ik wil graag beter worden in programmeren. Dus combineer ik deze drie onderwerpen om mezelf te verbeteren, en een prachtig PWS neer te zetten.

Onderzoek:
- Hoe visualiseren we hogere dimensionale objecten in lagere dimensies (Door middel van codering)?
    -Welke soorten “interdimensionale” projecties zijn er?
    -Welke informatie uit de hogere dimensies willen we behouden?
    -Welke andere toepassingen hebben hogere dimensies?
    -Hoe optimaliseren we het gebruik van hogere dimensionale datasets?

Hypothese:
Door middel van lineaire algebra is het mogelijk om hogere dimensies af te beelden naar een lagere dimensie.

Onderzoeksmethode:
Een deel van het onderzoek zal een literatuuronderzoek worden, maar een groot deel van het onderzoek zal ook plaatsvinden binnen het programma wat ik zal gaan schrijven. Dit programma zal hogere dimensionale fractalen kunnen gaan plotten, en vanaf hier kan ik deze fractalen projecteren naar lagere dimensies en daarmee onderzoek doen.

Begeleider communicatie:
Er is afgesproken dat ik Mevrouw Loeb een keer in de twee weken zal zien, uiteraard is contact daarbuiten ook mogelijk. Indien ik volgend jaar Wiskunde D of Wiskunde B van Mevrouw Loeb krijg is dat uiteraard ook een mogelijkheid voor contact.

Bronnen:
-3Blue1Brown: Een youtube kanaal toegewijd aan de visuele kant van wiskundige onderwerpen, heeft een complete introductie serie tot lineaire algebra en andere wiskundige onderwerpen.
-Wikipedia: Wikipedia is tegenwoordig betrouwbaarder dan veel officiële nieuws sites...
-Boeken:
-Python Crash Course: Een fast-paced introductie voor python om weer even bekend te worden met de taal.
-Geometrie van 4 dimensies: https://archive.org/details/geometryoffourdi032760mbp
-Een visuele introductie tot de vierde dimensie (Chris Mcmullen Ph D)
-Flatland (Edwin Abbot) Een roman over een tweedimensionale wereld met geometrische figuren als karakters. Behandelt veel interdimensionale perspectieven en omgangen.
-Python Optimalisatie Tricks

Logboek

Datum:	Werk:	Tijd (u):
Pre-13-7-19	Het idee van dit profielwerkstuk heb ik al sinds 5HAVO, en ik heb ook hier en daar naar de uitvoering van iets als dit gekeken.	~
13-7-19	Opzet PWS, Opzet PVA	3
14-7-19	Verdere uitwerking Opzet	5
15-7-19	Verdere uitwerking opzet	3
17-7-19	Begin Python onderzoek	6
18-7-19	Python onderzoek	5.5
20-7-19	Python Onderzoek, eerste Mandelbrot set gegenereerd	8
21-7-19	Python onderzoek, meerdere Mandelbrot sets gegenereerd	6.5
23-7-19	Python onderzoek, Mandelbrot_v2.py af.	5.5
28-7-19	Begin literatuuronderzoek	3.5
29-7-19	Setup van verslag structuur	1.5
7-8-19	Python Onderzoek voor geanimeerde fractals	2.5
8-8-19	Geanimeerde fractals, plus meer Python Onderzoek	1
12-8-19	Begin onderzoek naar OpenGL	5
13-8-19	Voortgezet onderzoek OpenGL	5
14-8-19	Verwerking van OpenGL onderzoek, met Python onderzoek	7
17-8-19	Literatuuronderzoek	1
18-8-19	Ontmoeting met Tristan voor bespreking programma	2.5
20-8-19	Start van mandelbrot_v3.py	4

21-8-19	mandelbrot_v3.py Gefinaliseerd, fractals kunnen nu in een kwestie van seconden worden gegenereerd.	6
22-8-19	Onderzoek naar generatie van Mandelbrot Set	3
1-9-19	Programmeer onderzoek, hoe kan een programma goed in elkaar worden gezet.	5
2-9-19	Vordering van programmeer onderzoek, mandelbrot_v3.py is nu netjes opgesteld	4.5
5-9-19	Voorbereiding Pitch + meer Python onderzoek	6
6-9-19	PWS Pitch, + verwerking van feedback van pitch	3
13-9-19	Verslag verder bouwen	2
14-9-19	Onderzoek naar projecties	3
15-9-19	Verwerking van onderzoek in verslag	4
18-9-19	Verwerking van onderzoek in verslag	4.5
20-9-19	Inleverdatum van eerste versie	1
28-9-19	Dag vol met programmeren, onderzoek naar Numpy, Python, OpenGL, en meer.	7.5
29-9	Tweede dag van programmeren, begin gemaakt aan OpenGL	7
5-10-19	Voortzetting van programmeren, en literatuuronderzoek	4.5
6-10-19	Voortzetting van literatuuronderzoek	5
15-10-19	Voortzetting van literatuuronderzoek	2.5
16-10-19	Programma barnsley_fern_generator.py gemaakt	2
20-10-19	Voortzetting van literatuuronderzoek	3
21-10-10	Voortzetting van literatuuronderzoek	2
24-10-19	Programma logistic_map.py gemaakt	2.5
25-10-19	Voortzetting van literatuuronderzoek	1.5
29-10-19	Samenvatting van al het programmeer onderzoek voor de toetsweek, zodat ik na de toetsweek weer gemakkelijk verder kan.	5
12-11-19	Toetsweek voorbij, maar alsnog erg druk. Klein onderzoek gedaan naar matrices	1
25-11-19	Fractals gegenereerd binnen mandelbrot_v3.py op nieuw bedachte manier, optimalisatie bonus.	4
30-11-19	Voortzetting van programmeeronderzoek	1.5
2-12-19	Bespreking van inleverdatum, 16-12	0.5
12-12-19	Voortzetting literatuuronderzoek	2.5
14-12-19	Uitwerking van eerste onderzoeksvragen, en uitwerking van verslag	4

16-12-19	Versie ingeleverd	0.5
17-12-19	Verdere uitwerking verslag	2
20-12-19	Programmeer onderzoek	3
21-12-19	Verdere uitwerking verslag	2.5
22-12-19	Verdere uitwerking verslag	4
27-12-19	Programmeer onderzoek	4
28-12-19	Verdere uitwerking verslag	5.5
2-1-20	Verdere uitwerking verslag	6
3-1-20	Verwerking van programmeeronderzoek in verslag	2
5-1-20	Verdere uitwerking verslag	2.5
7-1-20	Verdere uitwerking verslag	1
13-1-20	Verdere uitwerking verslag	2
20-1-20	Verdere uitwerking verslag	4
21-1-20	Vooronderzoek geconcludeerd, deelvragen beantwoord	6
23-1	Bespreking met docent over inkamping onderzoek	0.5
25-1-20	Verdere uitwerking verslag	7
27-1-20	Verdere uitwerking verslag	7
28-1-20	Verdere uitwerking verslag	6
29-1-20	Verslag klaar, wachtend op feedback van verschillende bronnen	5
30-1-20	Feedback verwerkt	6
31-1-20	Feedback verwerkt, en verslag gefinaliseerd	8
2-2-20	"Uitstel" gevraagd voor printen. Afgekeurd.	~
3-2-20	Ingeleverd.	~
	TOTAAL aantal uren:	258