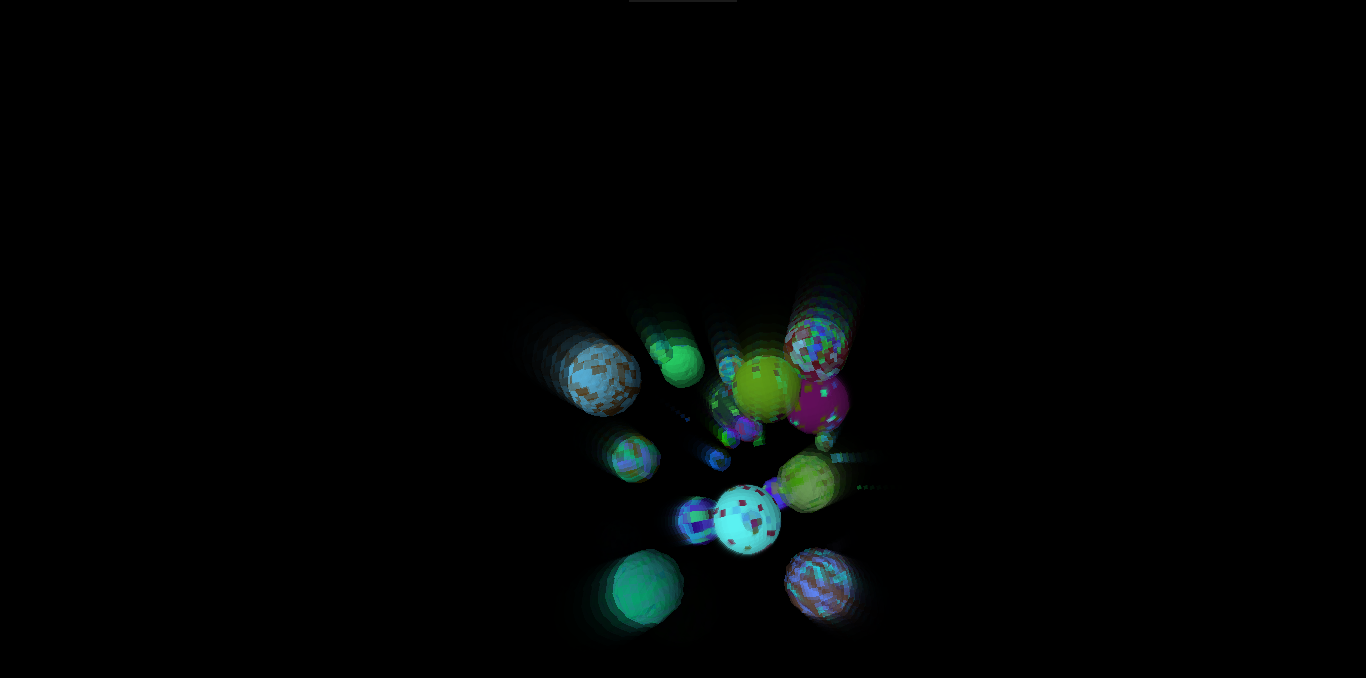
**Physics**



Jan Heikens, 5H6

Wartburg college Guido de Brès

Maart 2019

N. van Aken

Inhoud

[Inleiding  3](#_Toc2747919)

[Onderzoeksvraag  3](#_Toc2747920)

[Afbakening 3](#_Toc2747921)

[Deelvragen 3](#_Toc2747922)

[Onderzoeksmethode 3](#_Toc2747923)

[Benodigdheden: 3](#_Toc2747924)

[Planning 4](#_Toc2747925)

[Het maken van de 3D – Engine 5](#_Toc2747926)

[De basis van 3D tekenen 5](#_Toc2747927)

[De werking van je oog 5](#_Toc2747928)

[Van oog naar Engine 5](#_Toc2747929)

[Intcolor 11](#_Toc2747930)

[Het Heelal 11](#_Toc2747931)

[De zwaartekracht 11](#_Toc2747932)

[Een zwaartekrachtsimulator 12](#_Toc2747933)

[Een planeet 12](#_Toc2747934)

[Slot 13](#_Toc2747935)

[Links 14](#_Toc2747936)

[Literatuur 14](#_Toc2747937)

[Externe links 14](#_Toc2747938)

# Inleiding

Beste lezer,

Ik ben een programmeur en houd erg van algoritmes. Het mooie is dat je in de natuur heel veel formules en algoritmes kan vinden. Simulaties zijn dan ook best moeilijk te maken, omdat je nooit met alles rekening kunt houden. Een voorbeeld is de zwaartekracht. Het is redelijk eenvoudig te berekenen, maar het is moeilijker om een simulator te maken die de planeten in een baan rond de sterren laat gaan. Ik zou het leuk vinden om tijdens dit PWS ook een of meerdere simulators te maken. Dat zou een leuk resultaat zijn. Het gaat over het maken van een uitgebreid spel waarmee je met ruimteschepen door een procedureel gegenereerd universum kan reizen. Het wordt geprogrammeerd in C#, ik ben van plan er heel wat meer dan het minimum - 80 uur - in te steken.

Steeds pas ik mijn code aan, omdat ik steeds weer snellere manieren van coderen vind. Het verhaal zal in de loop der pagina’s steeds wat moeilijker worden.

# Onderzoeksvraag

Hoe kan de natuur nagebootst worden in een universum in een 3d simulatie?

# Deelvragen

* Hoe maak ik de 3D-Engine?
* Hoe laat ik een generator een heelal genereren op basis van een getal(“seed”)?
* Hoe maak ik een speler die door het heelal kan reizen?
* Hoe genereer ik het geluid?

## Afbakening

Een universum in een 3d simulatie:  
Een gesimuleerd universum, waarin je rond kan lopen en reizen, met planeten, manen, sterren en sterrenstelsels, zoals bijvoorbeeld in No Man’s Sky en Astroneer.

De natuur nabootsen:    
De natuur voor zover mogelijk in programmacode proberen te vangen, zoals de zwaartekracht toepassen, procedureel terrein en sterrenstelsels vormen, planten, mensen en dieren nabootsen.

Een ‘seed’ is een begin getal voor een random nummmer generator. Als je hetzelfde nummer erin stopt, komen er dezelfde nummers uit. Het heet ‘seed’ omdat het werkt als een zaadje: Al stop je hetzelfde zaadje in de grond, dan komt er dezelfde plant uit.

## Onderzoeksmethode

Ik voer mijn onderzoek uit door een simulatie te programmeren. De informatie hiervoor vind ik vooral door middel van Wikipedia en deel te nemen aan forums over programmeren. Daarnaast maak ik gebruik van Microsoft Visual Studio Community, een programma om programma’s te ontwikkelen.

## Benodigdheden

1. Laptop
2. MS Visual Studio
3. C#
4. Wikipedia
5. Google

# Planning

|  |  |
| --- | --- |
| Deelonderwerp | Data |
| **Fase 1** | **4-15 september** |
| Onderwerp/Onderzoeksvraag | 4-7 september |
| Bronnen zoeken | 10-14 september |
| **Fase 2** | **15 september-6 oktober** |
| Beginnen met het maken van de 3D-Engine,  Perlin noise, een zwaartekrachtsimulator en een gerandomiseerd universum. | 18 september |
| Fase 2 maken | 2-4 oktober |
| **Fase 3** | **6 oktober-22 december** |
| Verder gaan met het maken van dingen die ik nodig heb voor mijn onderzoek. | 6 oktober-13 oktober |
| Alle codes samenvoegen tot één geheel. | 15 oktober – 20 oktober |
| Optimaliseren | 23 oktober – 15 december |
| Het verslag verbeteren en in conceptvorm inleveren. | 17 december – 22 december |
| **Fase 4** | **22 december ’18-16 februari ‘19** |
| Het hele verslag (de verbeterde versie) inleveren |  |
| **Fase 5** | **16 februari-12 maart** |
| Presentatie maken | 16 februari – 12 maart |
| **Presentatieavond** | **12 maart** |
| **Afronding** | **Maart** |

# Het maken van de 3D – Engine

Als je een spel van het universum in 3d wilt maken, kan je twee dingen kiezen:

1: Een game-ontwerp programma gebruiken zoals Unity 3D.

2: Een eigen 3D-engine maken.

Zoals je waarschijnlijk veel zult merken, houd ik er erg van om alles zelf te maken. Ik kies dan ook voor de tweede optie. Ik zal je in dit hoofdstuk de basis leren van de 3D-Engine (spreek uit: ‘driedee endzjin’).

# De basis van 3D tekenen

## De werking van je oog

In je oog zitten allemaal receptoren die gevoelig zijn voor licht. Je hebt staafjes, die kunnen het best in de nacht zien en zijn groter, en kegeltjes, die overdag het meest in de weer zijn. Doordat de kegeltjes kleiner zijn, zitten ze verstopt achter de staafjes. Dat is maar goed ook, want overdag komt er veel meer licht je oog binnen en je zou al snel blind zijn!

De kegeltjes waar ik het net over had zijn verdeeld in drie soorten: rood-, groen- en blauwgevoelige kegeltjes. Die kegeltjes filteren dus hun kleur uit het licht. Die kleuren zijn ook de basis voor het lichtkleurenpallet. Als je nu denkt: ‘hè! Maar je mengt toch altijd kleuren met rood, geel en blauw?’, adviseer ik je het eens in Paint uit te proberen.

Omdat ons oog werkt als een Camera Obscura, een camera van vroeger die het zicht door een klein gaatje projecteerde op een lichtgevoelig achterwandje, zien wij in 2D.

Alles wat wij zien wordt omgedraaid. Maar het belangrijkste: Als je even redeneert, kom je erachter dat je de grootte van het object moet delen door de afstand, tenminste, als je een kijkwijdte van 90 graden hebt.

Dit project door zullen we deze hoek aanhouden als de kijkhoek.

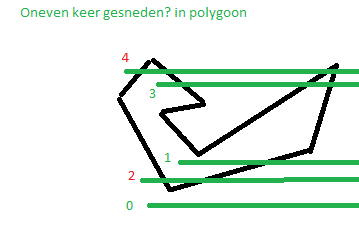
### Van oog naar Engine

Allemaal leuk en aardig, maar hoe zet je deze kennis om in programmeercode? Ik zal nu beschrijven hoe je dat doet. Degenen die niet geïnteresseerd zijn in programmeren raad ik aan deze deelparagraaf over te slaan.

Als eerste maak ik een class aan genaamd ‘OpenGL’. Ik noem deze class zo omdat hij in bepaalde aspecten op OpenGL, een 3D-Engine, gaat lijken. Nou *runt* OpenGL op een grafische kaart, die (Letterlijk!) honderd keer sneller gaat dan een CPU, doe ik niet eens de moeite die ongelofelijke snelheid te overtreffen. Voordat ik verder ga, vertel ik over een 3D-Engine.

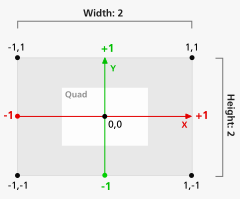
Een gemiddelde 3D-Engine tekent alles uit driehoeken; Ga voor jezelf eens na hoe je eenvoudige vormen in driehoeken omzet. Neem een vierkant: die kan je omzetten in twee driehoeken. Neem een kubus: die kan je omzetten in twaalf driehoeken, namelijk voor alle vlakken twee driehoeken. Alleen bollingen zijn onmogelijk te maken. Je kunt ze wel nabootsen, door met zoveel mogelijk precisie driehoeken in de vorm van een rondje te tekenen. Daarom moet de engine als eerste in staat zijn driehoeken te tekenen.

Deze class heeft een basis class: de BitmapLocker. Die class zet de bitmap eenvoudigweg om in bytes als hij ‘gelockt’ wordt en weer terug in een bitmap als hij ‘ge-unlockt’ wordt. Een bitmap wordt opgeslagen in een byte array, met vier bytes voor iedere pixel: Blue van de hoeveelheid blauw, G van de hoeveelheid groen, R van de hoeveelheid R en A van de hoeveelheid Alpha. Die Alpha-waarde indiceert de doorzichtigheid. Al is de Alpha 255, is die pixel niet doorzichtig en al is de Alpha nul, dan is die pixel volledig doorzichtig. Het aantal bytes dat een Bitmap gebruikt is dus de breedte \* hoogte \* 4 bytes per pixel.

Als je een polygoon[[1]](#footnote-2) wilt tekenen, kan je voor iedere pixel een eindeloze lijn naar rechts tekenen en checken hoe vaak die lijn kruist met de lijnen tussen de punten van die polygoon. Als dat aantal oneven is, is het punt in de polygoon.

Maar nog sneller is om voor iedere rij te kijken wat de linkerlijn van de driehoek is en wat de rechterlijn. Als je dat weet trek je van de x van de linkerlijn op die hoogte een lijn naar de x van de rechterlijn op die hoogte. Er kan maar één linker- en rechterlijn zijn bij een driehoek.

Nu we het algoritme hebben om de driehoeken te tekenen, kunnen we 3D- driehoeken gaan omzetten naar 2D – driehoeken. Voor iedere driehoek berekenen we eerst de DX, verschil in x-as, DY en DZ. Dan roteren we alles over het XY-vlak en daarna over het YZ-vlak en doen de Y \* -1 om de afstand tot de camera in de Y- as te kunnen meten. Als de camera over de x- en y- as nul graden gedraaid is, kijkt de camera dus in de min-richting over de y-as.



Om de coördinaten te converteren naar punten op het scherm, pakken we eerst het maximum van de breedte en hoogte van het beeldscherm. Als dus het beeldscherm breder is dan hij hoog is, pakken we dus de breedte. Dit is de vermenigvuldigingsfactor. Omdat alles wat we willen zien in bereik -1 tot 1(de kijkwijdte) zal liggen, delen we de factor door twee. Daarna vermenigvuldigen we alle punten met die factor en delen ze door de afstand. Als laatste tellen we bij de x de helft van de breedte en bij de y de helft van de hoogte op. Als we dit met alle driehoeken die voor het scherm zitten gedaan hebben, kunnen we het nog een schaduw geven. Als een van de drie lijnen over de positieve x-as naar beneden gaat, geven we de driehoek een schaduw.

Maar hoe maak je een mooie schaduw? Dat is heel eenvoudig, je vermenigvuldigt een kleur gewoon met een schaduwkleur. Laten we zeggen dat de schaduw roodachtig is van de zonsondergang. Dan geven we de schaduwkleur

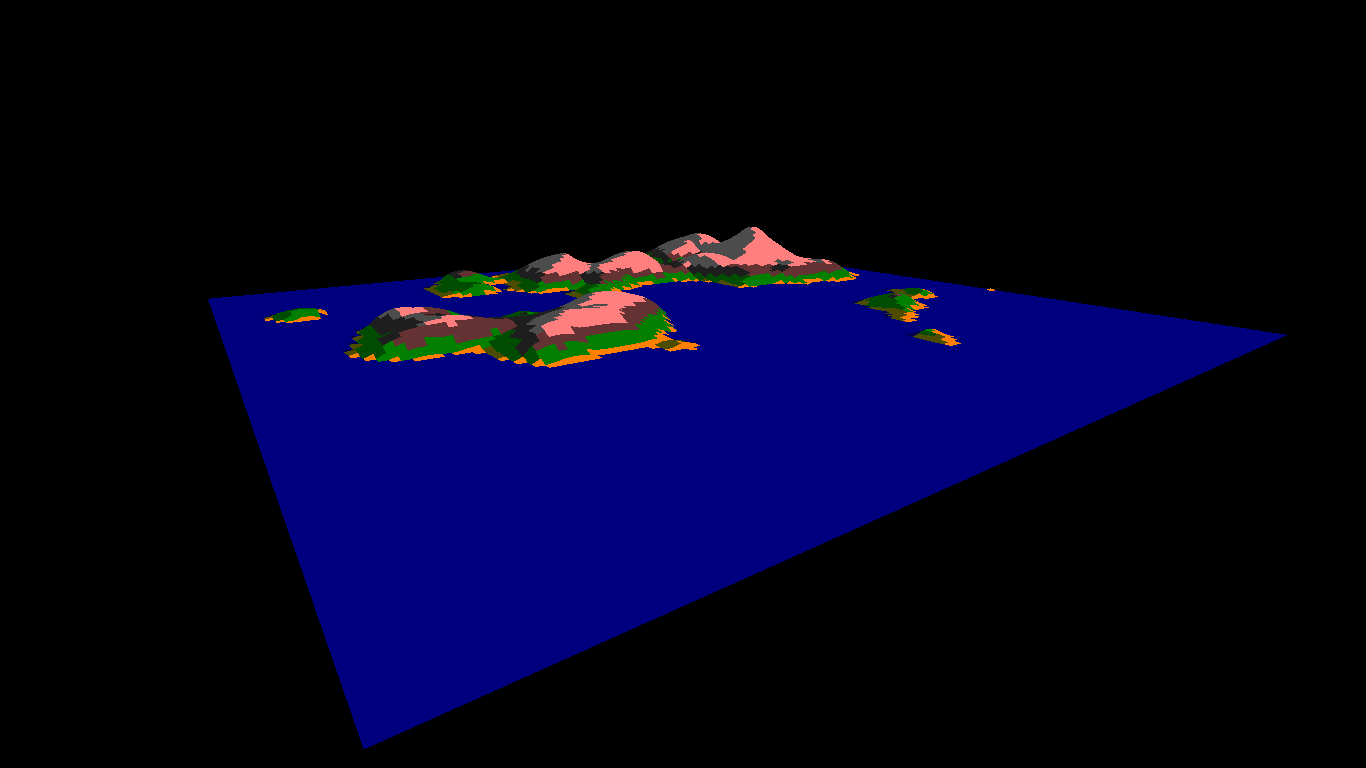
Rood: \*0,5

Groen: \*0

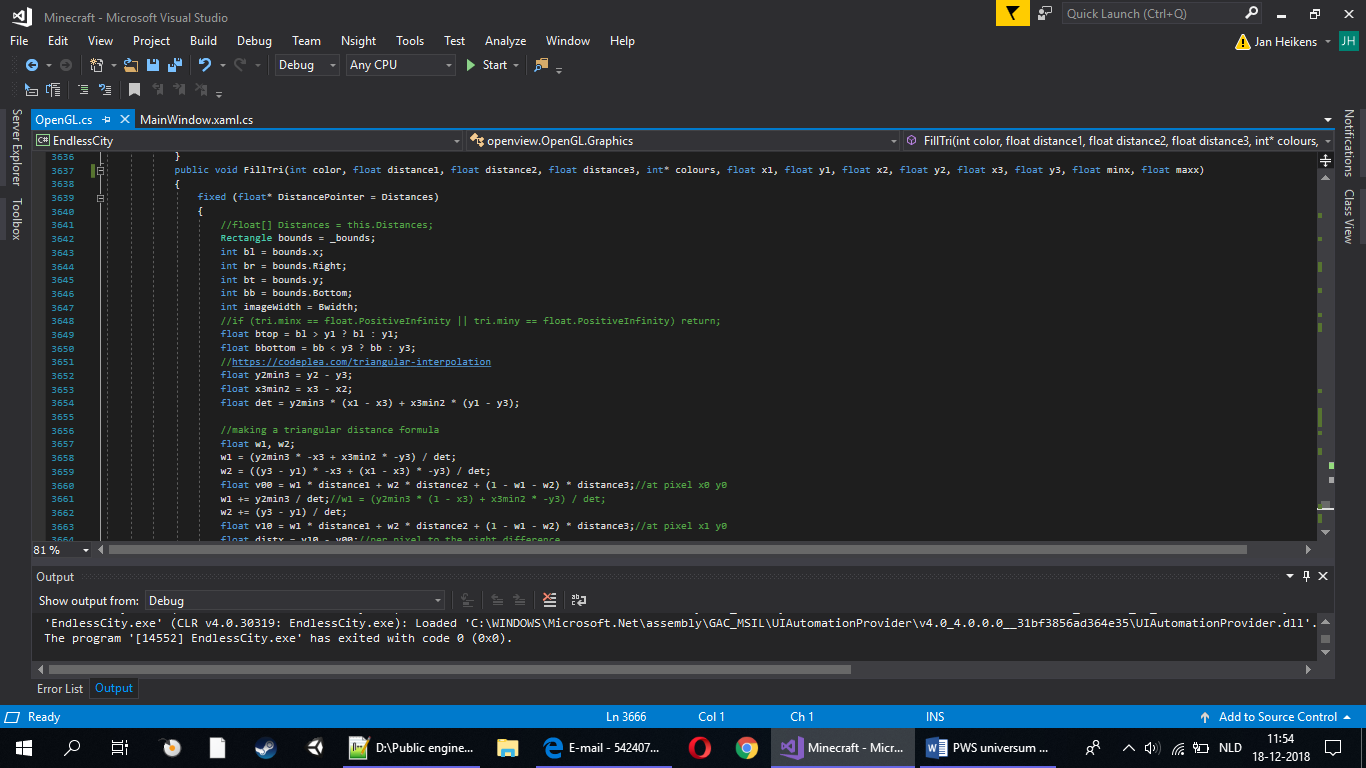
Blauw: \*0,2

Zoals je ziet geef ik de schaduw ook wat blauw mee, omdat een zonsondergang meestal paarsig is. Als we zwart vermenigvuldigen met de schaduw, blijft het zwart, want nul keer iets is altijd nul. Als we wit hiermee vermenigvuldigen krijgen we de schaduwkleur, want je kan wit als ‘1’ opvatten en een keer iets is altijd iets.

Dit is een van de *outputs* van mijn 3D- engine.



(87 ms om te maken) en daarmee bedoel ik niet de engine, maar dit plaatje!



Dit is het begin van de functie om een driehoek op te vullen. Zoals je ziet halen we eerst alle variabelen van het object op. Elke keer als je die ophaalt, kost het meer rekenkracht dan het ophalen van de lokale variabelen. Daarom slaan we deze objectvariabelen op in lokale variabelen.

Om de driehoek binnen de grenzen te houden kijken we of de driehoek over de grens van het tekenvlak gaat. Als dat zo is, beperken we de driehoek tot de grens door de variabele btop de waarde bl te geven.

Uitleg variabele namen:

Btop = *bounds* top y

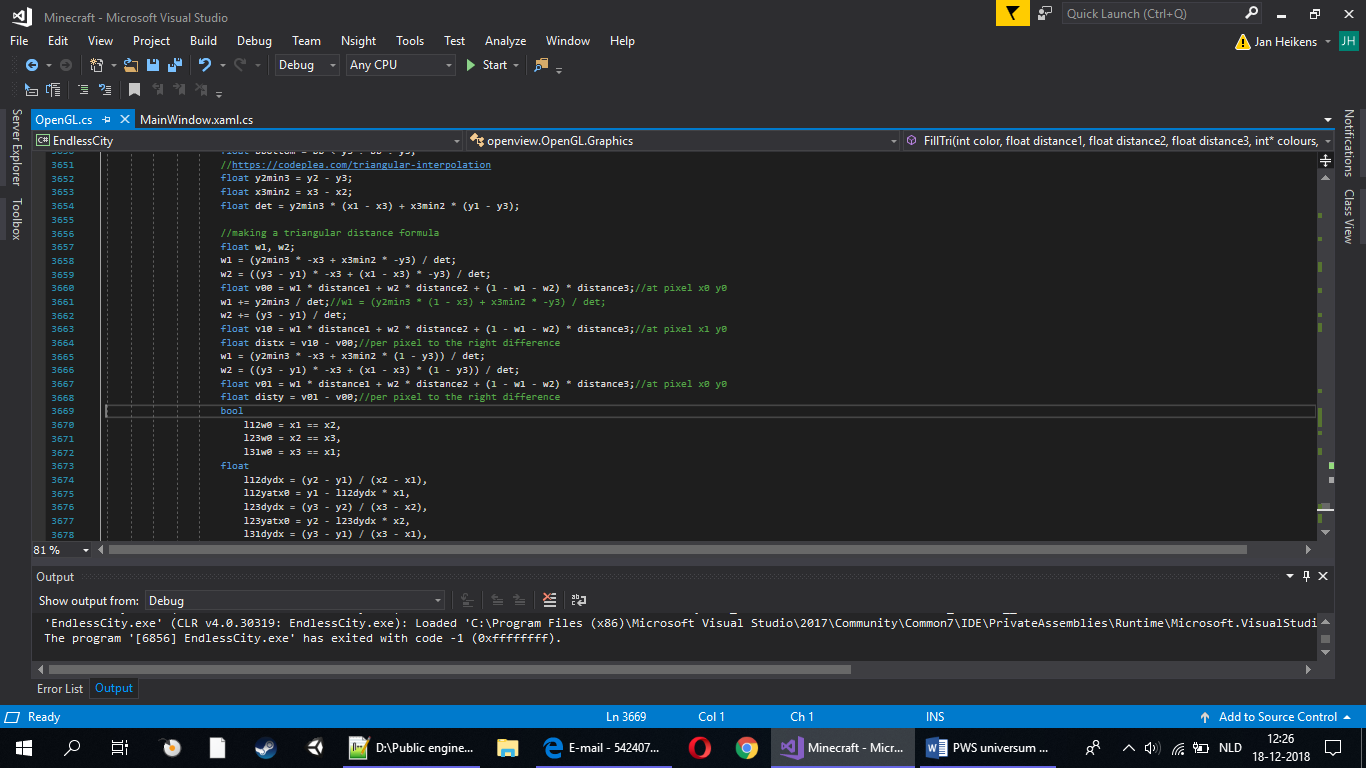
Bleft = *bounds* left x

Bbottom = *bounds* bottom y

Brigt = *bounds* right x

Voor de driehoek. Hier gaat hij tekenen.

Bt, bl, bb en br hebben dezelfde waarden, alleen dan voor het tekenvlak.



De link uit dit stuk code: <https://codeplea.com/triangular-interpolation>

De tweede ‘right difference’ moet ‘bottom difference’ zijn.

Voor iedere pixel ‘op’ een driehoek bereken ik de afstand doormiddel van een driehoeksformule: dist = distx \* px + disty \* py + v00 oftewel z = ax + bx + c-vorm.

Waar v00 = distantie op de linksboven pixel,

Distx = de cumulatieve afstandsvermenigvuldiger met x (iedere x komt er *distx* bij),

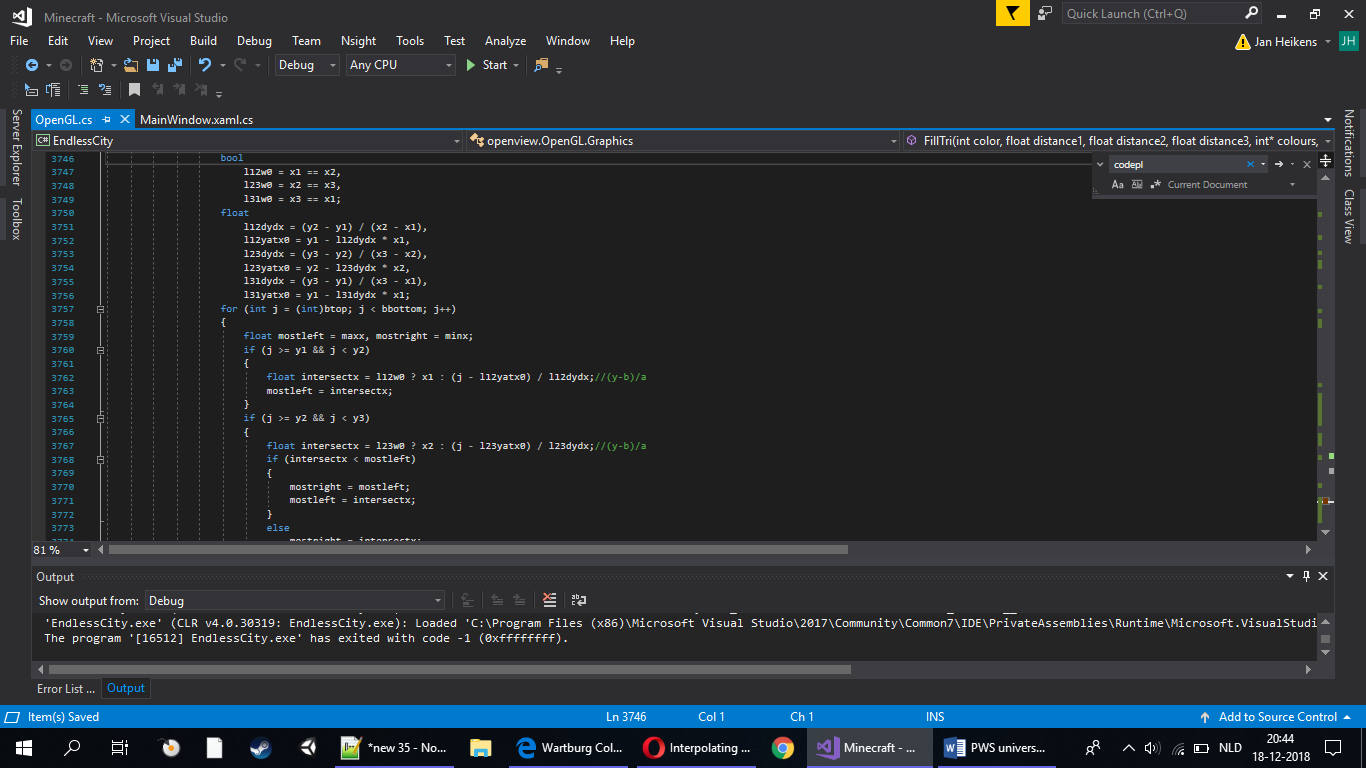
Disty = de cumulatieve afstandsvermenigvuldiger met y (iedere y komt er *disty* bij),

Px = de x-coördinaat van de pixel en

Py = de y-coördinaat van de pixel.

Ik bereken de waarden op drie plaatsen om de waarden van de formule te weten, namelijk op x0y0, x1y0 en x0y1. Hieruit volgt dat *distx* = de waarde op x1y0-x0y0 en *disty* = de waarde op x0y1 – x0y0 moet zijn.

Hierna begint hij met de rijen. Voor iedere rij berekent hij wat de meest linkse pixel is die hij moet vullen en de meest rechtse. Hij zorgt ervoor dat hij ook binnen de x-*bounds* tekent.

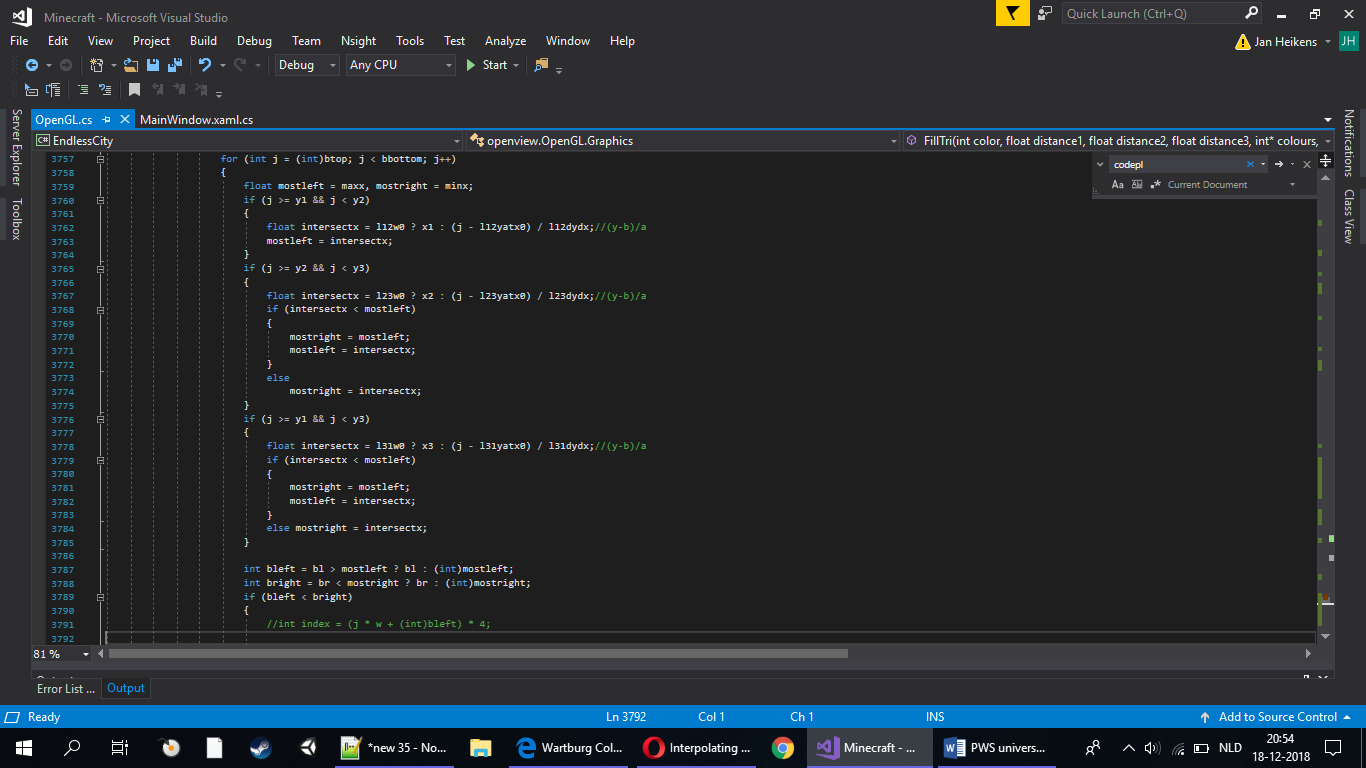


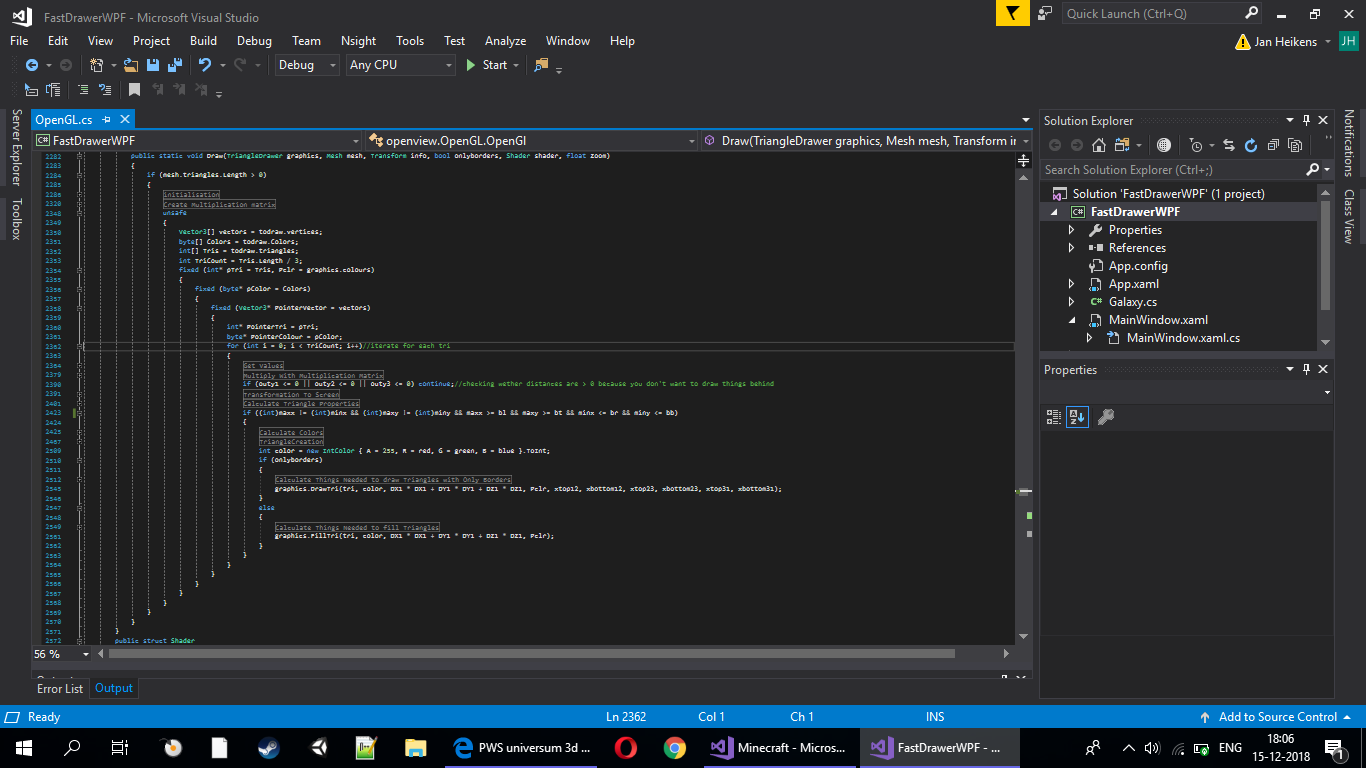
Hier worden variabelen aangemaakt die een andere functie maken.

L##w0 geeft aan of de breedte van de lijn nul is, om problemen met *infinity*(/0) te voorkomen.

L##dydx = de a van de lineaire functie y = ax + b

L##yatX0 = de b.



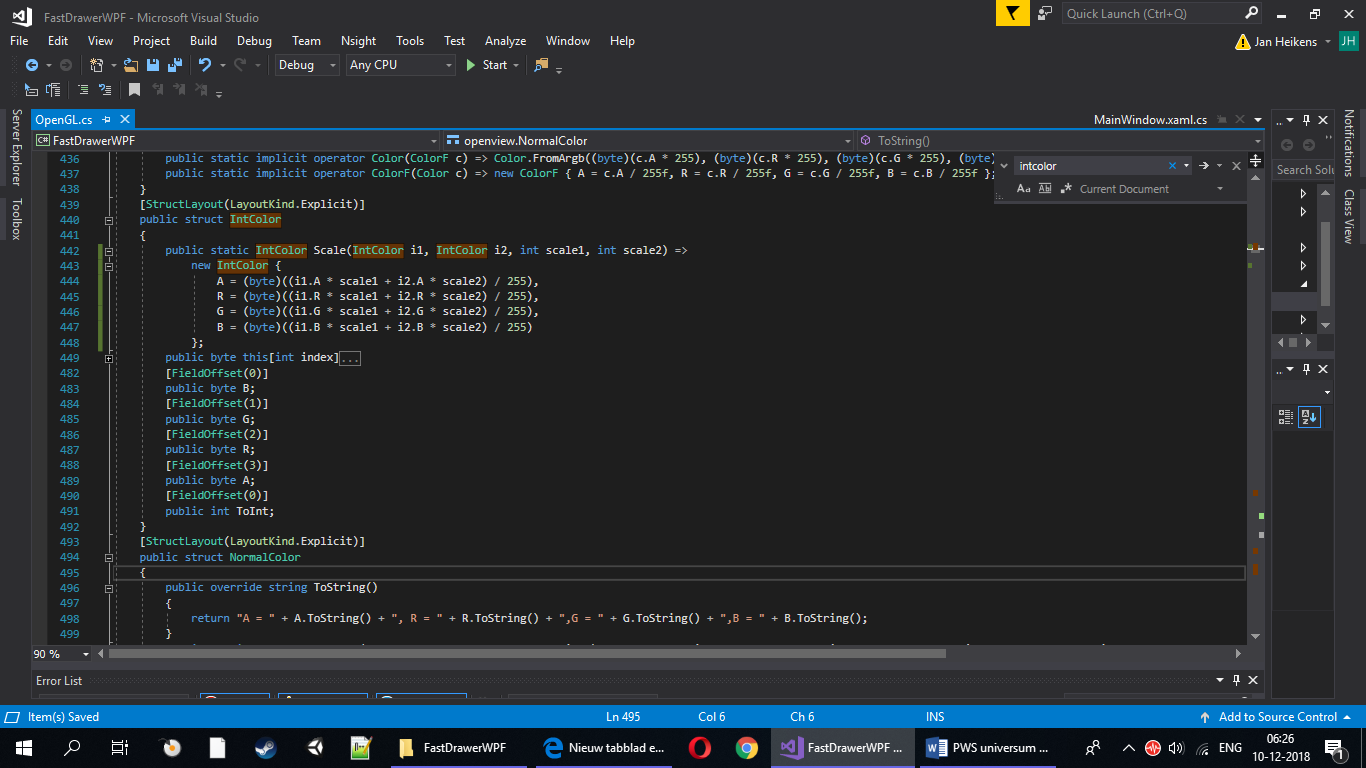


Dit is de functie die gebruikt wordt om de driehoeken te laten omzetten en te tekenen. Ik heb alles dichtgeklapt,

Omdat het te lang is om in een keer te laten zien.

### Intcolor

De 3D engine die dit tekent maakt gebruik van een *graphics* object, gespecialiseerd om driehoeken te tekenen, lijnen te tekenen, drie hoeken op te vullen, rechthoeken op te vullen, plaatjes te tekenen, met of zonder een bepaalde *alpha* waarde.



De *Struct Intcolor* maakt gebruik van *Fieldoffsets*; een int zit nu op dezelfde plaats in het geheugen als de 4 bytes, zodat je razendsnel bytes uit een int kan halen.

Hiervoor hebben we het nodig, razendsnel de 4 bytes van een kleur om te zetten in één integer-waarde. De snelste methode is dan om de bytes op dezelfde plaats als de integer op te slaan.

Dat kan je bepalen met *Fieldoffsets*. Zoals u kunt zien, wordt de eerste byte op 0 opgeslagen, de tweede op 1, de derde op 2 en de vierde op 3. We slaan de int op in 0 tot en met 3.

Pixels hebben in plaats van 4 afzonderlijke bytes dus een *integer*-waarde. Dit kunnen we versnellen met *Pointers*.

# Het heelal

In het spel moeten we met ons ruimteschip kunnen reizen door het heelal. Omdat je, als je van het ene sterrenstelsel naar het andere wilt gaan, enorm ver moet reizen, moet je dus gigantisch snel gaan. Als je zo snel gaat zie je niet veel en daarom zal je waarschijnlijk tijdens de reis één boel licht zien. Als je in je ruimteschip zit, verander je niet van plaats, dus zie je het schip altijd precies hetzelfde, behalve dan wat je door de ramen ziet. Misschien laat ik het programma een ‘screenshot’ van het schip van binnen geven die nooit verandert. Alleen op de ramen tekent hij dan in 3d.

## De zwaartekracht

De formule voor zwaartekracht is

G \* (m1 \* m2)/(d2)

Daarvoor heb je dus een gravitatieconstante nodig:

public const float gravConst = 0.1f;//(6.674×10−11 N · (m/kg)2)

Als je dit in code omzet krijg je

float newton = gravConst \* m1 \* m2 / (distance \* distance) \* Speed;//newton per frame

Zoals je kan zien, heb ik er een extra factor bij gedaan, de Speed. Dat geeft aan hoeveel eenheden er verstrijken voor de volgende berekening. Hoe meer tijd er dus verstrijkt, hoe meer er aan de planeet is getrokken. Al wil je de kracht in snelheid omzetten om een object te bewegen, deel je de kracht door de massa van een object:

float newton = (gravConst \* m1 \* m2 / (distance \* distance) \* Speed)/m1;//velocity per frame

Je kan de twee “m1’tjes” weghalen. Dan krijg je

float newton = gravConst \* m2 / (distance \* distance) \* Speed;//velocity per frame

## Een zwaartekrachtsimulator

Om de formule uit te testen hebben we een zwaartekrachtsimulator nodig. Maar wat is een zwaartekrachtsimulator?

In een goede simulator:

* Kan je sterrenstelsels maken, door planeten in een baan rond andere planeten te brengen.
* Wordt het gevisualiseerd met 3D *graphics*.
* Kan je planeten volgen, ‘met ze mee reizen’.
* Kunnen planeten botsen en heeft dat impact op de planeten.

## Een planeet

Om een planeet te maken moeten we als 1e terrein kunnen genereren door Perlin Noise (noise betekent vibratie, schommelingen in het heen en weer gaan). Perlin komt van de bedenker Ken Perlin. Hij is de 1e die een procedureel terrein wist te genereren wat er heel realistisch uitzag. Het concept is heel eenvoudig; het is gebaseerd op een tweedimensionale *array* van willekeurige getallen.

Als je dus de hoogte van een bepaald punt met coördinaten wil weten, waarbij de x en de y een heel getal zijn, pak je de waarde ban het gegenereerde getal uit de *array*. Als je dit doet krijg je heel erg ruig terrein. Het is namelijk niet op elkaar afgestemd, maar random. Maar nu komt het mooiste. Als je waarden tussen de hele getallen wilt weten, kun je er tussen interpoleren door middel van de cosinus. In het engels heet het *lerp.* De functie *lerp* retourneert hij het 1e getal als de x 0 is; als de x 1 is retourneert hij het 2e getal. Als het 0,5 is, retourneert hij het gemiddelde van het 1e en 2e getal. Dit betekent dat de formule kan worden neergezet:

*Y = v1+ (v2-v1) \* x*

Dit is een lineaire formule.

Om de sinus te verwerken moeten we eerst deze formule aanroepen voordat we het in de functie lerp stoppen.

*Y = sin(x \* pi/2)*

# Slot

Lang heb ik aan dit profielwerkstuk gewerkt. Ik vond het heel leuk om te doen. Ik heb geleerd om onder andere een 3d-engine te maken. Jammer genoeg raakte het project een beetje *out of date*: Ik heb nu al veel meer geleerd over 3d-engines en maak nu spellen met een veel betere beeldkwaliteit en een waanzinnige *performance*. Als ik dit werkstuk over zou moeten doen met even veel kennis als toen, had ik waarschijnlijk hetzelfde gedaan. Ik zou wel beter op de documentatie letten, want aan het begin van mijn PWS ging het daar nog niet al te best mee. Bij een vervolgonderzoek zou ik een nieuwe engine maken met *multisampling*. Ook zou ik het leuk vinden een project over zwarte gaten te maken, waarin ik molecuulsgewijs zwarte gaten simuleer. Moleculen geven licht af als ze tegen elkaar botsen. Ik zou er dan voor zorgen dat er fotonen zijn et cetera. Jammer genoeg is dit nog veel te zwaar om te draaien voor de computers uit deze tijd. Maar wie weet!

Hier is opgesomd wat ik behaald heb:

Behaald:

* + De 3D engine, zoals beschreven in mijn verslag.
  + De planeten

Half behaald:

* + Een planeet volgen:

Je kan jezelf op een planeet zetten, maar dat werkt niet goed.

* + Botsende planeten:

Ook dit werkt niet goed. Als ze samenvoegen, komt er een heel rare planeet uit. Daarom heb ik dit uit gezet.

Niet behaald:

* + Het verwerken van geluid op basis van afstand tot de speler.

# Links

## Literatuur

Code plea.(3 jaar geleden). *Interpoleren met Barycentrische coördinaten.* Geraadpleegd in 2018, van <https://codeplea.com/triangular-interpolation>

Red blob games. (26 Apr 2018). *Making maps with noise functions*. Geraadpleegd op 5 oktober 2018, van <https://www.redblobgames.com/maps/terrain-from-noise/>

Github.(z.d). *Mersenne priemgetal random nummer generator*. Geraadpleegd in 2018, van <https://gist.github.com/adamveld12/6c0350d1cfd2da449dc6>

Reddit. (1 jaar geleden). *No man’s sky procedurele dieren.* Geraadpleegd op  8 October 2018, van <https://www.reddit.com/r/NoMansSkyTheGame/comments/60wykl/proper_procedural_generation_of_animals/>

Youtube. (z.d). *Procedural terrain generation*. Geraadpleegd op 5 October 2018, van <https://www.youtube.com/playlist?list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3>,

<https://github.com/SebLague/Procedural-Landmass-Generation>

Gamedev.(7 jaar geleden). *Snijpunten lijn met bol*. Geraadpleegd in 2018, van <https://gamedev.stackexchange.com/questions/20815/most-efficient-bounding-sphere-vs-ray-collision-algorithms>

## Externe links

Astroneer official reveal trailer:

<https://www.youtube.com/watch?v=cv7xbHkLdGw>

No man’s sky gameplay trailer:

<https://www.youtube.com/watch?v=nLtmEjqzg7M>

1. Een polygoon is een veelhoek. [↑](#footnote-ref-2)