**Verslag PGT:**

**Procedurally Generated Terrain**

Naam: Jonathan Vercammen

Klas: 3ICT

Schooljaar: 2022-2023

Inhoud

[1 Inleiding 1](#_Toc123997032)

[2 Noisefuncties 1](#_Toc123997033)

[2.1 Introductie 1](#_Toc123997034)

[2.2 Noise 1](#_Toc123997035)

[2.3 Random Number Generator (RNG) 2](#_Toc123997036)

[2.4 Random noise 3](#_Toc123997037)

[2.5 Value noise 3](#_Toc123997038)

[2.5.1 Werking 4](#_Toc123997039)

[2.5.2 Eigenschappen 5](#_Toc123997040)

[2.6 Perlin noise 6](#_Toc123997041)

[2.6.1 Werking 6](#_Toc123997042)

[2.6.2 Eigenschappen 7](#_Toc123997043)

[2.7 Simplex noise 8](#_Toc123997044)

[2.7.1 Werking 8](#_Toc123997045)

[2.7.2 Eigenschappen 9](#_Toc123997046)

[3 Fractal noise 10](#_Toc123997047)

[3.1 Parameters / Begrippen 10](#_Toc123997048)

[4 Procedurally Generated Terrain 13](#_Toc123997049)

[4.1 Introductie 13](#_Toc123997050)

[4.2 3D rendering 14](#_Toc123997051)

[4.3 De kubus 14](#_Toc123997052)

[4.4 Eenvoudig terrein genereren 15](#_Toc123997053)

[4.5 Noise integreren in terrein 17](#_Toc123997054)

[4.6 Optimalisatie 1: chunks 18](#_Toc123997055)

[4.7 Optimalisatie 2: custom mesh renderer 19](#_Toc123997056)

[4.8 Optimalisatie 3: multi-threading 19](#_Toc123997057)

[4.9 Problemen 20](#_Toc123997058)

[5 Besluit 20](#_Toc123997059)

[6 Codefragmenten 20](#_Toc123997060)

[7 Figurenlijst 21](#_Toc123997061)

[8 Bronvermelding 23](#_Toc123997062)

[9 Bijlagen 25](#_Toc123997063)

# Inleiding

Dit onderzoek verdiept zich in het thema van terreingeneratie. Eerst komt een introductie over wat noise is en de verschillende noisefuncties. Daarna volgt de uitleg over terreingeneratie en een voorbeeld gemaakt in Unity.

# Noisefuncties

## Introductie

Willekeurigheid is overal in de wereld maar toch blijft het zo’n abstract begrip. Wanneer is iets random? Is het ontstaan van het universum of het gooien van een dobbeslteen random en hoe kunnen computers dit simuleren als ze enkel instructies kunnen volgen? Vooral de laatste vraag piekte de interesse van Ken Perlin, een professor van de New York-universiteit. Perlin vond namelijk dat computergegenereerde beelden niet organisch genoeg waren.

Tijdens het maken van de Walt Disney-film *Tron (1982)* bedacht hij een algoritme dat organische vormen kon maken. Enkele jaren later in 1997 kreeg Perlin een academische prijs van AMPAS, de Academy of Motion Picture Arts and Sciences, en kreeg het algroritme de toepasselijke naam Perlin Noise. (Wikipedia, Perlin Noise, 2022)

## Noise

Noise of ruis heeft verschillende betekenissen. Enerzijds duidt het op een storend geluid. Anderzijds betekent het een visuele storing. Denk bv. aan de sneeuw van een tv of het gekras van een radio. Voor een signaal betekent noise een willekeurige afwisseling in sterkte of snelheid.

Hoofdzakelijk bestaan er twee soorten noise namelijk value noise en gradient noise. Gradient noise heeft veel varianten. Enkele voorbeelden zijn Simplex noise, Perlin noise, simulation noise en wavelet noise. Er zijn ook speciale soorten noise zoals Worley noise of Fractal noise.

De volgende hoofdstukken gaan verder in op de meest gekende noisefuncties en hun werking.

## Random Number Generator (RNG)

Elke noisefunctie steunt op random getallen. Value noise bijvoorbeeld gebruikt deze getallen om rasterpunten op te vullen. Maar hoe kunnen computers getallen genereren? Dit is waar random number generators (RNG) van pas komen.

Computers genereren geen echte random getallen maar pseudorandom getallen. Dit zijn getallen die willekeurig lijken maar opgebouwd zijn door een vaste formule. Het grootste voordeel hiervan is dat het zeer snel werkt. Ook is er maar een kleine hoeveelheid geheugen nodig. Het nadeel hieraan is dat een formule altijd hetzelfde resultaat zal teruggeven. Om meer variatie te geven, maakt een RNG gebruik van een ‘seed’. Dit is een cijfer die bepaalt hoe de generator getallen berekent.

Een voorbeeld van een random number generator is de lineaire-congruentiegenerator. Deze heeft als formule: waarbij

|  |
| --- |
| X0­­­: seed / startwaarde |
| m: modulus, ofwel de rest na deling |
| a: vermenigvuldigingsfactor |
| c­­­: toename |

Hieronder een voorbeeld met parameters X0­­­=5, m=6, a=2 en c­­­=8.

Hier is meteen een probleem zichtbaar. Na voldoende iteraties ontstaat er zich een patroon. Parameterkeuze is bij deze generator heel belangrijk. Wanneer de uitkomst kleiner is dan tien dan zullen er in totaal maximum tien iteraties gebeuren zonder dubbele getallen. Daarom is het belangrijk om de parameters zo groot mogelijk te houden. Zo is de kans kleiner dat hetzelfde getal twee keer voorkomt.

Om een voorbeeld te geven, Java gebruikt voor haar Random()-methode volgende parameters: m= 2­­­­48, a= 25.214.903.917 en c­­­= 11. (Wikipedia, Linear congruential generator, 2022)

## Random noise

De simpelste vorm van noise is random noise. Random noise bestaat uit een n-dimensioneel raster waarin elk punt een willeukeurige waarde krijgt tussen nul en één, respectievelijk zwart en wit. In de figuur hiernaast is een voorbeeld te zien van random noise in twee dimensies.

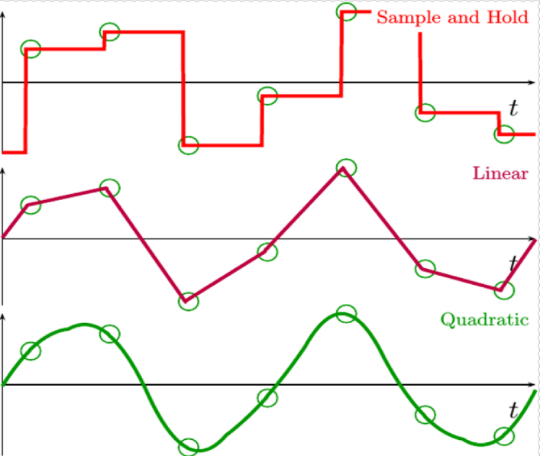
Het is duidelijk zichtbaar dat dit geen natuurlijk resultaat oplevert maar dat is ook niet het doel van random noise. Enkele toepassingen van random noise, beter bekent als white noise, zijn het maskeren van storende geluiden en in de geluidssector om versterkers te testen of audiosynthese. (Wikipedia, White noise, 2022)

Figuur 1 Een 2D raster met white noise

## Value noise

Net zoals random noise bestaat value noise uit een n-dimensioneel raster van willekeurige punten. Het verschil is dat value noise alle tussenwaardes interpoleert door te kijken naar de dichtstbijzijnde rasterpunten. In één dimensie betekent dit een interpolatie tussen de twee dichtstbijzijnde rasterpunten. In 2D, tussen de vier dichtstbijzijnde rasterpunten. Hiernaast staat een voorbeeld van value noise. Figuur 2

Figuur 2 Een 2D raster met value noise

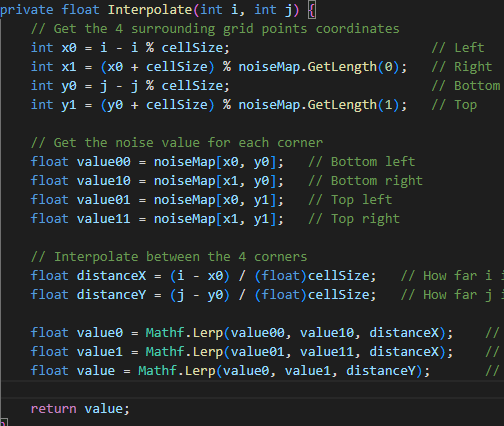
[](https://zipcpu.com/dsp/2018/03/30/quadratic.html)Er bestaan verschillende technieken om te interpoleren, meer hierover in volgend hoofdstuk. De meest gebruikte zijn linear en quadratisch. Lineair geeft snel resultaat maar ziet er vaak blokkerig uit. Quadratisch geeft het beste resultaat maar heeft veel rekenkracht nodig. Op onderstaande afbeelding is een voorbeeld zichtbaar van lineare en quadratische interpolatie. (Archer, 2011)

Figuur 3 Verschil tussen lineare en quadratische interpolatie

### Werking

Value noise start met een n-dimensioneel raster. In onderstaand voorbeeld staat de werking uitgelegd van value noise in twee dimensies met lineare interpolatie.

Stel eerst een 2D raster op en vul elk punt met een willekeurige waarde tussen nul en één **Error! Reference source not found.**. Voor elke waarde tussen deze punten, bereken de geïnterpoleerde waarde van de vier omliggende rasterpunten. Onderstaand codefragment toont een manier om deze waarde te berekenen voor een punt (i , j).

Het eerste blok code zoekt de coördinaten van de cel waarin het punt zich bevindt Figuur 5. Daarmee worden de waardes van elk hoekpunt opgevraagt Figuur 6. Vervolgens berekent de code de positie van het punt in de huidige cel en zet die om in procent Figuur 7. Ten slotte wordt interpolatie drie keer toegepast. Figuur 8

Codefragment 1 Interpolatie tussen twee punten i en j

De eerste keer voor de onderste twee punten, de tweede keer voor de bovenste twee punten en de laatste keer om verticaal te interpoleren.

Deze techniek heet ‘bi-lineare interpolatie’.

|  |  |  |  |
| --- | --- | --- | --- |
| Figuur 4 Raster met vier punten | Figuur 5 Celcoördinaten zoeken | Figuur 6 Waardes van hoekpunten zoeken | Figuur 7 Positie in cel zoeken |
| Figuur 8 Bi-lineare interpolatie |  |  |  |

### Eigenschappen

* Zeer snel voor één laag value noise maar schaalt niet goed met meerdere lagen.
* Hoeft noisewaarden niet op te slaan in het geheugen.
* Kwaliteit is sterk afhankelijk van de interpolatietechniek.

## Perlin noise

Perlin noise is de meest gebruikte noisefunctie voor game development. Het is snel en geeft een natuurlijk resultaat.

Perlin noise is uitgevonden in 1982 door Ken Perlin. Later in 2001 verbeterde hij zijn algoritme dat nu gekend is als simplex noise, meer hierover in volgend hoodstuk.

Figuur 9 Een 2D raster met Perlin noise

In tegenstelling tot value noise gebruikt Perlin noise een raster van eenheidsvectoren i.p.v. punten. Deze vectoren duwen alle tussenliggende waardes weg van de oorsprong.

Onderstaande tekst legt de werking uit van Perlin noise in twee dimensies.

### Werking

Perlin noise werkt met twee lijsten: een lijst met vectoren in willekeurige richting en een lijst met random natuurlijke getallen, de permutatielijst. Deze lijst bestaat uit n getallen tussen 0 en n – 1. Dit vervangt de random number generator die nodig is bij value noise.

De eerste stap is dezelfde als bij value noise namelijk: stel een 2D raster op van willekeurige eenheidsvectoren, kies een punt en zoek zijn omliggende rasterpunten. Zoek daarna voor elk van deze punten hun bijhorende vector uit de vectorlijst met een hashfunctie Figuur 10. Een voorbeeld hiervan is: waarbij ‘p’ de permutatietabel is. Bereken daarna voor elk hoekpunt de afstandsvector naar het punt in de cel door het verschil te nemen van beide punten Figuur 11. De laatste stap is de interpolant berekenen van deze vier waardes. Dit gebeurt op dezelfde manier als bij value noise namelijk met lineare interpolatie.

|  |  |  |  |
| --- | --- | --- | --- |
| Figuur 10 gradient vectoren opzoeken | Figuur 11 afstandsvectoren berekenen |  |  |

### Eigenschappen

* Is een stuk trager dan value noise.
* Schaalt slecht bij meerdere dimensies
* Gebruikt een beetje geheugen om de vector- en permutatielijst op te slaan.
* Geeft een mooi resultaat.
* Afhankelijk van de implementatie kan Perlin noise herhalen.

## Simplex noise

Simplex noise is de verbeterde versie van Perlin noise die ook is gemaakt door Perlin zelf. Het grootste probleem met Perlin noise is dat het heel traag werkt in meerdere dimensies. De oplossing die Perlin hiervoor heeft bedacht, is om te werken met een ander raster en weg te stappen van gewone vierkanten.

### Werking

In de meetkunde zijn er een aantal manieren om een multidimensionele figuur weer te geven. Eén daarvan is de hyperkubus. De meest bekende varianten zijn het vierkant in twee dimensies en de kubus in 3D.

Een tweede manier is met een simplex. Simplices zijn de famillie van de driehoek en de piramide. In onderstaande afbeeldingen staan de meest gekende hyperkubussen en simplices nog eens op een rijtje.

|  |  |  |
| --- | --- | --- |
| Figuur 12 2D hyperkubus, beter bekent als een vierkant | Figuur 13 3D hyperkubus, beter bekent als een kubus | Figuur 14 4D hyperkubus, beter bekent als een tesseract |

|  |  |  |
| --- | --- | --- |
| Figuur 15 2D simplex, ofwel een driehoek | Figuur 16 3D simplex, ofwel een tetraëder | Figuur 17 4D simplex, ofwel een pentachoron |

Een voordeel van simplices is dat er veel minder hoekpunten zijn om te berekenen. Bij Perlin noise is het aantal hoekpunten in n dimensies gelijk aan 2n. Bij simplex noise is dit maar n+1. Een bijkomend voordeel is dat lineare interpolatie volledig wegvalt. Simplex noise gebruikt een radiale dempingsfunctie om de tussenwaardes te berekenen.

De grootste moeilijkheid bij simplex noise is het vinden van de omliggende rasterpunten. De meest gebruikte techniek is om eerst alle rasterpunten te projecteren op een 2D vlak. De berekening van de omliggende hoekpuntent is vrij complex en zal niet worden besproken in dit verslag. Nadat alle hoekpunten bekend zijn, is de rest analoog met Perlin noise. Bereken daarna voor elk hoekpunt het dot product van de afstandsvector en de vector in het hoekpunt. De laatste stap gebruikt een dempingsfunctie om de uiteindelijk waarde te bekomen. Deze vervangt de interpolatie die nodig is bij Perlin noise en is ook een stuk sneller. (Archer, 2011)

### Eigenschappen

* Sneller dan Perlin noise
* Schaalt zeer goed met meerdere dimensies
* Gebruikt een beetje geheugen voor de permutatietabel en vectoren
* Geeft een beter resultaat dan Perlin noise

# Fractal noise

Fractal noise, of layered noise, is een veelgebruikte techniek om noise meer ‘diepte’ te geven. De naam fractal noise komt van het meetkundig begrip ‘fractaal’. Een fractaal is een figuur die volledig zelfsymetrisch is. Dit wil zeggen dat op elke plaats een patroon zichtbaar is dat zichzelf continu herhaald tot in het oneindig. Het bekendste voorbeeld hiervan is de Mandelbrot, zie Figuur 19.

|  |  |  |
| --- | --- | --- |
| [The Amazing World of Fractals | Be Loud! - A Yizzam Blog](https://blog.yizzam.com/the-amazing-world-of-fractals/)  Figuur 18 Voorbeeld van een fractaal | [How to Render a Fractal, Fast](https://blog.bede.io/how-to-render-a-fractal-fast/)  Figuur 19 Mandelbrot set | [Sierpiński triangle - Wikipedia](https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle)  Figuur 20 Driehoek van Sierpiński |

Wat heeft dit nu allemaal te maken met noise? Fractal noise werkt met verschillende lagen. Elke laag is gebaseerd op de vorige en heeft meer details. Hoe deze lagen veranderen t.o.v. elkaar is te bepalen door enkele parameters.

## Parameters / Begrippen

Het eerste belangrijk begrip is de amplitude. De amplitude van een functie geeft weer wat de maxima en minima van die functie zijn. Voor noise specifiek betekent dit hoe groot, of klein, de noisewaarde op elk punt kan zijn. Hieronder enkele voorbeelden met amplitudes 20, 100 en 300.

|  |  |  |
| --- | --- | --- |
| Figuur 21 Fractal noise met amplitude 20 | Figuur 22 Fractal noise met amplitude 100 | Figuur 23 Fractal noise met amplitude 300 |

Het tweede begrip is de frequentie. Deze parameter bepaalt hoe snel noisewaardes kunnen veranderen. Een hoge frequentie geeft veel meer details weer dan een lage frequentie. Onderstaande afbeeldingen tonen dit effect aan.

|  |  |  |
| --- | --- | --- |
| Figuur 24 Fractal noise met frequentie 1 | Figuur 25 Fractal noise met frequentie 0.5 | Figuur 26 Fractal noise met frequentie 2 |

De derde parameter is het aantal octaven. Neen, het zijn geen muzieknoten maar wel het aantal iteraties noise dat samengevoegd wordt. Hoe meer octaven, hoe meer details er zich zullen vormen. Let wel op want dit is een zware bewerking dus hou deze parameter klein. Op volgende afbeeldingen staat het verschil uitgebeeld tussen 1, 3, 6 en 10 octaven. De meeste toepassingen gebruiken vier octaven noise omdat dit het beste resultaat geeft voor de rekentijd.

|  |  |  |
| --- | --- | --- |
| Figuur 27 Fractal noise met 1 octaaf | Figuur 28 Fractal noise met 3 octaven | Figuur 29 Fractal noise met 6 octaven |
| Figuur 30 Fractal noise met 10 octaven |  |  |

De laatste twee parameters zijn iets complexer en hebben een verband met de voorgaande parameters. Het vierde begrip is lacunariteit en geeft weer hoe snel de frequentie stijgt per octaaf of per iteratie. Deze parameter is meestal gelijk aan twee.

|  |  |  |
| --- | --- | --- |
| Figuur 31 Fractal noise met lacunariteit 2 | Figuur 32 Fractal noise met lacunariteit 1 | Figuur 33 Fractal noise met lacunariteit 3 |

Het vijfde en laatste begrip is de persistentie en bepaalt hoeveel de amplitude verminderd per octaaf. De standaardwaarde is 0,5 maar andere waarden zijn ook mogelijk.

|  |  |  |
| --- | --- | --- |
| Figuur 34 Fractal noise met persistentie 0,5 | Figuur 35 Fractal noise met persistentie 1 | Figuur 36 Fractal noise met persistentie 0,1 |

# Procedurally Generated Terrain

Eindelijk zitten we aan het hoofdonderwerp van dit verslag: procedurally generated terrain. Dit hoofdstuk zal tal van informatie bevatten over hoe PGT werkt met een werkende demo die geschreven is in de Unity game-engine. Vanaf nu zal ook de tekst in persoonlijke stijl geschreven zijn.

## Introductie

Eerst en vooral: wat is procedurally generated terrain (PGT)? Om dat te weten is het belangrijk om eerst uit te leggen wat procedural, of procedureel, betekent.

Procedural content is when the game will generate content either during the running of a game, or at the initial load. Be it new loot drops, game spaces, or enemy types, there are a lot of options here

*- Josh Bycer* (Bycer, 2018)

M.a.w. procedurele content betekent dat de game volledig nieuwe content genereert tijdens of voor het spelen. Verwar dit zeker niet met random generation. Dit is wanneer een game content maakt op basis van vooropgestelde onderdelen. Procedural generation heeft als groot voordeel dat developers niet alles zelf moeten maken op voorhand. Procedurally generated terrain betekent dus terrein dat is gegenereerd door een computer.

## 3D rendering

Er bestaan hoofdzakelijk twee verschillende technieken om een 3D-object op het scherm te tonen. De eerste manier is met voxels. Een voxel is het best vergelijkbaar met een 3D pixel.

De tweede manier is met polygons. Een polygon is een gesloten 2D figuur die bestaat uit meerdere hoekpunten, of vertices, die met elkaar verbonden zijn door zijden.

|  |  |  |
| --- | --- | --- |
| Figuur 37 Wereld gemaakt met voxels | Figuur 38 Verschil tussen voxels en polygons | Figuur 39 Polygon met alle onderdelen aangeduid |

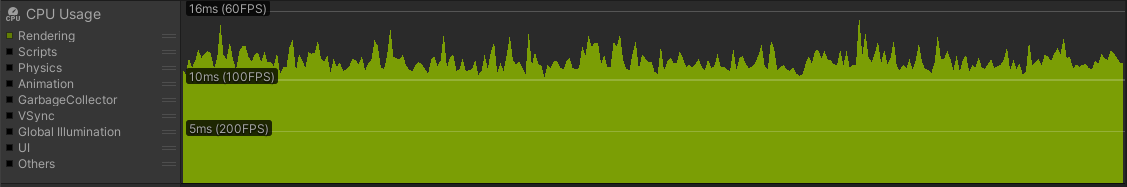
Een groot voordeel van voxels is dat ze makkelijk bewerkbaar zijn. Het is eenvoudig om voxels toe te voegen of te verwijderen van een object. Dit is een stuk moeilijker bij polygons maar die hebben dan weer als voordeel dat ze heel getailleerde figuren kunnen opslaan zonder al te veel polygonen te gebruiken.

De demo in dit project zal gebruik maken van polygons.

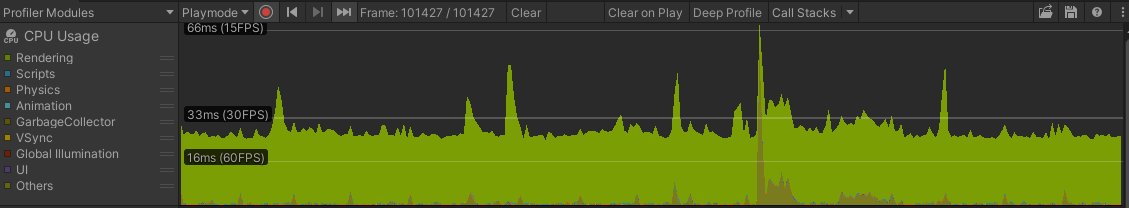
## De kubus

De eerste stap naar succesvol terrein is een kubus. In Unity bestaan er enkele standaardobjecten zoals een kubus, bol, cylinder en capsule. Je zou dan denken dat het heel eenvoudig is om terrein te genereren. Kopiëer gewoon de standaard kubus en klaar. Hoewel dit leuk klinkt, is er toch een groot probleem mee namelijk performantie. Onderstaande screenshots tonen aan hoe de performantie drastisch verslechtert naarmate het aantal blokken toeneemt.

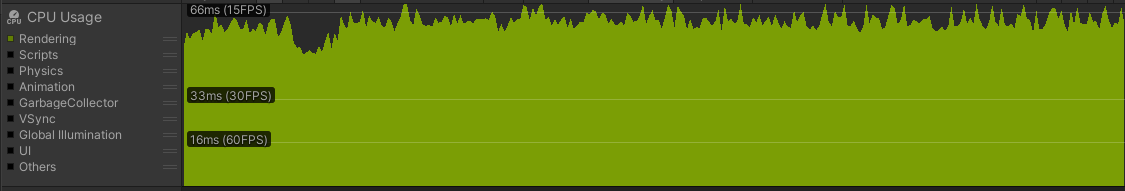
Figuur 40 Standaardobjecten van Unity



Figuur 41 Performantie bij ~5000 blokken



Figuur 42 Performantie bij ~10 000 blokken

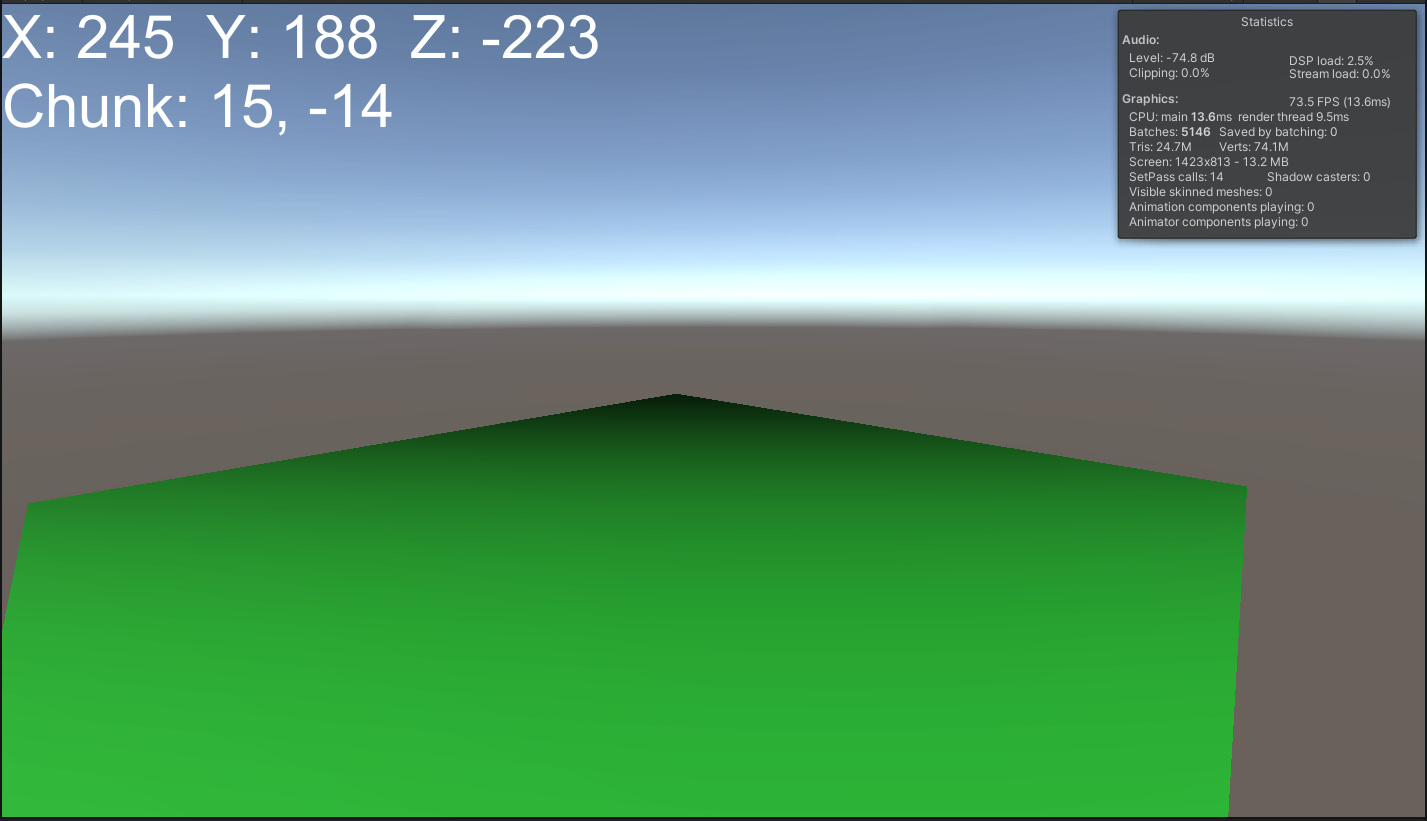


Figuur 43 Performantie bij ~20 000 blokken

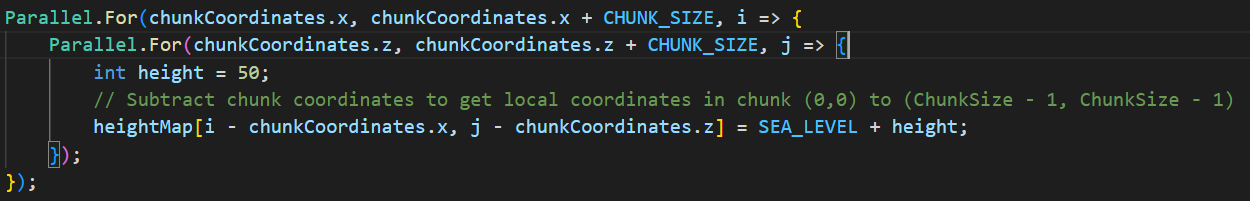
Dit is niet echt bruikbaar voor grote stukken terrein maar voorlopig volstaat deze techniek.

## Eenvoudig terrein genereren

De simpelste vorm van terrein is een lege vlakte. Dit is heel eenvoudig in code te vertalen.

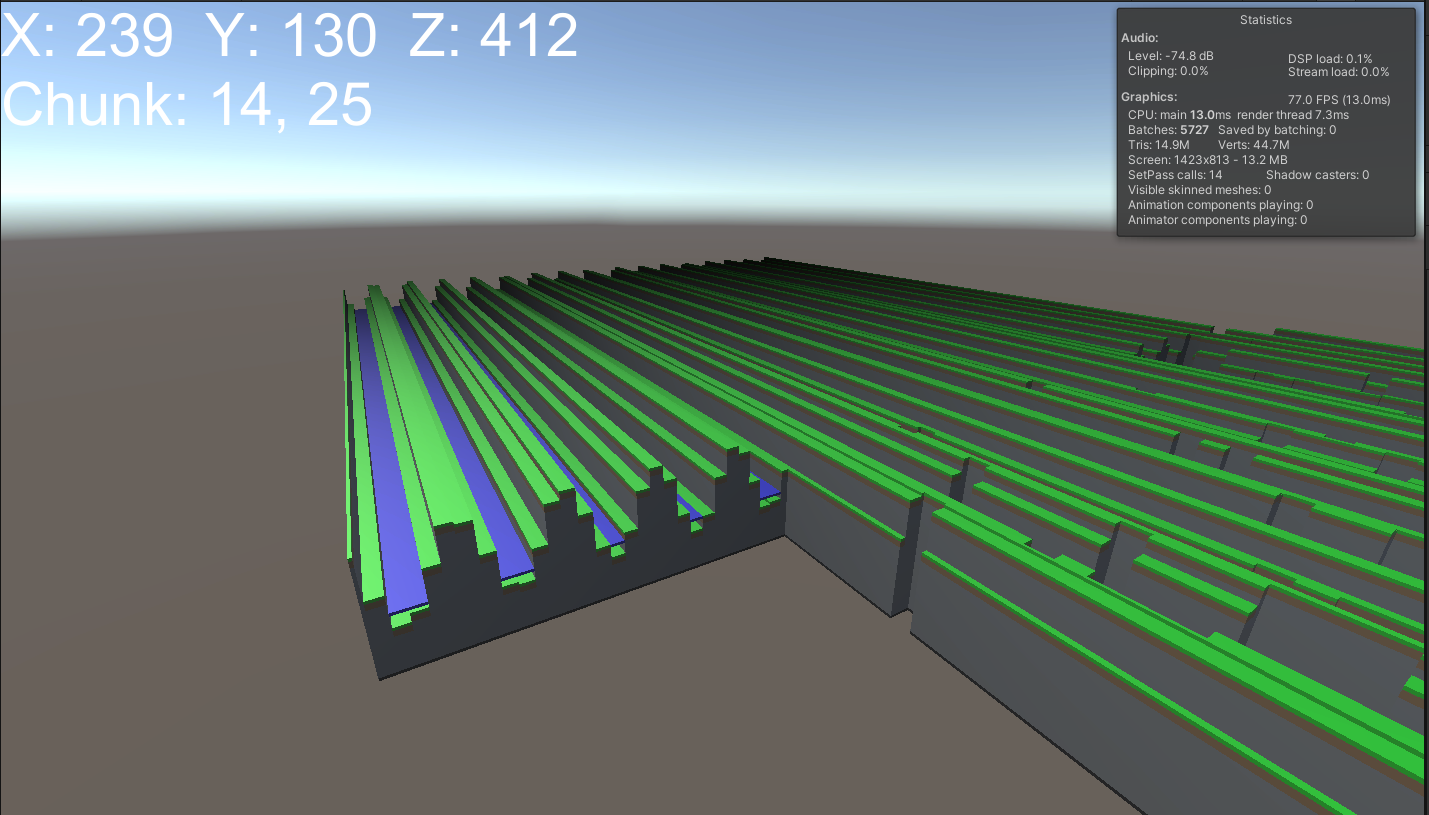


Figuur 44 Vlak terrein

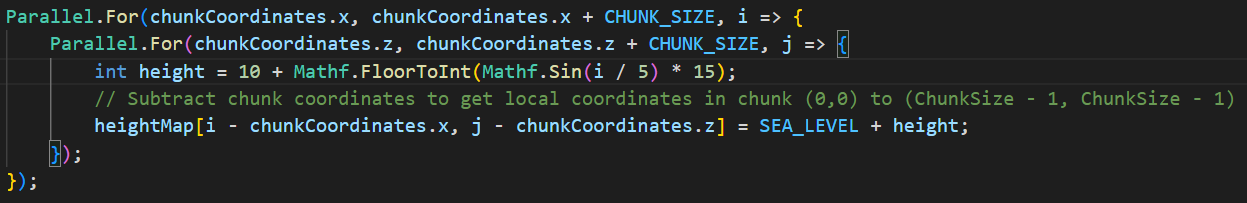


Codefragment 2 Code voor vlak terrein

Een andere manier is om gebruik te maken van wiskunde functies. Neem bijvoorbeeld de sinusfunctie.



Figuur 45 Sinusvormig terrein



Codefragment 3 Code voor sinusvormig terrein

Dit is in principe ook procedurally generated terrain maar het is niet heel interessant om in rond te lopen.

## Noise integreren in terrein

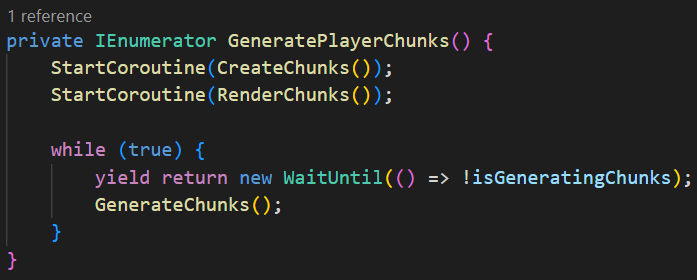
De volgende stap is om een noisemap te gebruiken voor de terreinhoogte. In dit geval gebruik ik drie octaven simplex noise die als input de x- en z-coördinaat neemt en als output de hoogte teruggeeft.

Met deze aanpak is het terrein direct veel natuurlijker. Het belangrijkste om een realistisch terrein te genereren is de keuze van de parameters. Bij een klein aantal lagen noise is het terrein vrij glad en heeft het weinig detail. Hieronder een voorbeeld met één laag noise en vier lagen noise.

|  |  |
| --- | --- |
| Figuur 46 Terrein met één laag noise | Figuur 47 Terrein met vier lagen noise |

Het is vrij duidelijk dat de rechtste figuur veel natuurlijker terrein geeft dan de linkse. Dus waarom niet gewoon tien lagen noise gebruiken? De eerste foto duurde ongeveer 15 seconden om volledig te genereren. Bij de tweede foto duurde dit rond de veertig seconden. Het berekenen van een noisemap vergt vrij veel rekenkracht. Naarmate het aantal lagen noise toeneemt, verhoogt ook de rekentijd.

## Optimalisatie 1: chunks

De meest eenvoudige manier om de performantie te verbeteren is door simpelweg minder blokken tegelijk op het scherm te renderen. Dit kunnen we doen door het terrein op te splitsen in kleinere delen of ‘chunks’. Al deze chunks kunnen dan apart gegenereerd en gerenderd worden. Dit is veel beter dan het volledig terrein in één keer te renderen. Ook bestaat nu de mogelijkheid om in parallel te werken.

Codefragment 4 GeneratePlayerChunks methode

De code hiervoor ziet er zo uit:

Er zijn twee Coroutines die zich bezighouden met het aanmaken van chunks en het renderen van chunks. Daarnaast is er een loop die continu chunks genereert.

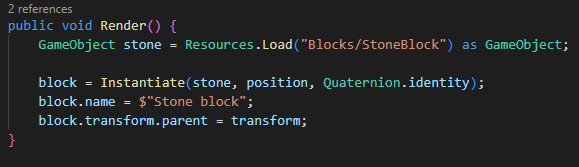
De CreateChunks Coroutine haalt een lijst op van lege chunkposities op met behulp van de asynchrone functie GetEmptyChunkPositions(). Deze lijst bevat alle chunks rondom de speler die nog niet gemaakt zijn. Vervolgens wordt voor elke positie in die lijst een nieuwe chunk aangemaakt en toegevoegd aan de chunks dictionary.

De GenerateChunks()-methode genereert de heightmaps en meshes voor alle chunks rondom de speler. Het gebruikt een achtergrondtaak om de lijst van chunks te krijgen die al gemaakt zijn en dus klaar zijn om verder gegenereerd te worden. Als er geen chunks zijn om te genereren, stopt de methode. Anders start het opnieuw een achtergrondtaak om de heightmaps van de chunks te genereren en daarna ook hun meshes. Ten slotte wordt de ‘isGeneratingChunks’ boolean op false gezet om aan te geven dat het genereren voltooid is.

De RenderChunks Coroutine loopt door de chunks dictionary en roept voor elke chunk die gegenereerd is de UploadMesh() functie aan. Hiermee wordt de mesh van de chunk naar de graphics card gestuurd en wordt ze ook getoont op het scherm. Bij een fout tijdens het renderen, verwijdert de chunk zichzelf en zal hij opnieuw worden gegenereerd.

## Optimalisatie 2: custom mesh renderer

De tweede optimalisatie heeft alles te maken met de GPU. Nu wordt er voor elke positie onder de hoogte een blok geplaatst. Zoals eerder vermeld is deze manier niet echt efficiënt. Zeker niet met de standaard Unity kubus omdat alle zijden altijd gerenderd worden, ongeacht ze nu zichtbaar zijn of niet. Zo ziet de code er nu uit gebruikmakend van de Unity kubus.



Codefragment 5 Oude manier van terrein renderen

Om dit probleem op te lossen creëer ik mijn eigen mesh met de methode GenerateOptimizedMesh(). Deze methode is te vinden in de bijlage als Codefragment 6.

Nog eens een kleine opfrissing: een mesh is een verzameling van vertices (hoekpunten), lijnen en driehoeken die samen een mesh vormen. De methode begint door de status van de chunk op ‘Generating’ te zetten. Zo wordt vermeden dat een chunk iets anders gaat beginnen doen zoals renderen vooraleer hij gegenereerd is. Vervolgens worden een aantal variabelen geïnitialiseerd, zoals een kleur en een transparantie voor de mesh. Daarna worden twee arrays geïnitialiseerd: ‘faceVertices’ en ‘faceUVs’. Deze arrays zullen later worden gebruikt om de vertices en UVs van elke face op te slaan. Ook is een array genaamd ‘neighbors’ geïnitialiseerd, deze bevat alle blokken die zich rond de huidige blok bevinden.

Daarna wordt er geïtereerd over alle blokken van de chunk. Voor elk blok wordt de positie en het blokobject opgeslagen in de variabelen ‘blockPos’ en ‘block’. Ook de kleur van het blok is opgeslagen. Vervolgens wordt de array ‘neighbors’ gevuld met de methode GetNeighbours().

Vervolgens wordt er een loop uitgevoerd voor elke zijde van de blok. Binnen deze loop wordt er nog een loop uitgevoerd over elke vertex van de huidige zijde. In deze loop worden de vertices en UVs van de huidige zijde opgeslagen in de arrays ‘faceVertices’ en ‘faceUVs’. Dit gebeurt door gebruik te maken van twee hardcoded arrays, ‘voxelVertices’ en ‘voxelUVs’, die vaste vertices en UVs bevatten voor een zijde.

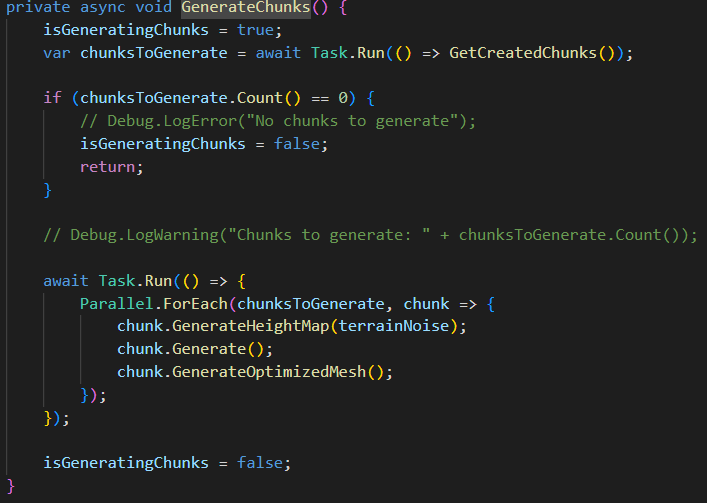
De vertices van de huidige zijde worden opgehaald door te zoeken in de array ‘voxelVertices’ met behulp van de ‘voxelVertexIndex’-array, die de index bevat van elke vertex voor elke zijde. Vervolgens worden deze vertices opgeslagen in de ‘faceVertices’-array door er de positie van de huidige blok bij op te tellen. Daarna wordt de blok dat naast de huidige face ligt, opgeslagen in de variabele "neighbor". Als deze buur leeg of transparant is dan wordt de huidige face getekend.

Als alle blokken van de chunk zijn gecontroleerd dan wordt alle data van vertices, UVs, colors en triangles opgslaan in een struct MeshData.

## Optimalisatie 3: multi-threading

Door het terrein op te splitsen, is het nu mogelijk om meerdere chunks tegelijk te genereren in parallel. Jammer genoeg kan niet alles in een aparte thread gedaan worden. De meeste Unity-methodes moeten in de main thread gebeuren. Gelukkig heeft Unity hiervoor een handige oplossing voor namelijk Coroutines.

Een coroutine in Unity is een functie die in staat is om een reeks instructies uit te voeren in meerdere frames. Een coroutine kan bijvoorbeeld worden gebruikt om opdrachten te onderbreken en later voort te zetten, wat handig kan zijn wanneer er wachttijden nodig zijn. Het grootste voordeel is dat alles nog steeds gebeurt op de main thread maar nu gewoon over meerdere frames wordt verdeelt.

Voor andere zaken die niet op de main thread moeten werken volstaat een gewone async-await structuur met C# Tasks. In de code zijn er veel zaken die parallel gebeuren maar de belangrijkste is het genereren van chunks. Dit is ook de zwaarste taak van heel het programma. Hieronder een uitleg over hoe deze methode werkt.

De methode begint met het instellen van de variabele ‘isGeneratingChunks’ op true. Zo kan de methode maar één keer tegelijk draaien.

Daarna vraagt hij alle chunks op die al gemaakt zijn. De await zorgt ervoor dat de thread niet blokkeert terwijl hij wacht op het resultaat.

Codefragment 6 GenerateChunks methode

Vervolgens wordt gecontroleerd of de lijst ‘chunksToGenerate’ leeg is. Als dit zo is dan wordt de functie afgebroken. Als er wel chunks zijn om te genereren, wordt er een nieuwe taak gestart die deze chunks parallelliseert en de blokken en mesh genereert voor elk van deze chunks. Ten slotte wordt de variabele ‘isGeneratingChunks’ terug op false ingesteld om aan te geven dat de iteratie is voltooid. Zo kan de volgende batch van chunks worden gegenereerd.

# Besluit

# Codefragmenten

[Codefragment 1 Interpolatie tussen twee punten i en j 4](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997064)

[Codefragment 2 Code voor vlak terrein 15](#_Toc123997065)

[Codefragment 3 Code voor sinusvormig terrein 16](#_Toc123997066)

[Codefragment 4 GeneratePlayerChunks methode 18](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997067)

[Codefragment 5 Oude manier van terrein renderen 19](#_Toc123997068)

[Codefragment 6 GenerateOptimizedMesh methode 25](#_Toc123997069)

# Figurenlijst

[Figuur 1 Een 2D raster met white noise 3](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997070)

[Figuur 2 Een 2D raster met value noise 3](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997071)

[Figuur 3 Verschil tussen lineare en quadratische interpolatie 3](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997072)

[Figuur 4 Raster met vier punten 4](#_Toc123997073)

[Figuur 5 Celcoördinaten zoeken 4](#_Toc123997074)

[Figuur 6 Waardes van hoekpunten zoeken 4](#_Toc123997075)

[Figuur 7 Positie in cel zoeken 4](#_Toc123997076)

[Figuur 8 Bi-lineare interpolatie 4](#_Toc123997077)

[Figuur 9 Een 2D raster met Perlin noise 6](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997078)

[Figuur 10 gradient vectoren opzoeken 6](#_Toc123997079)

[Figuur 11 afstandsvectoren berekenen 6](#_Toc123997080)

[Figuur 12 2D hyperkubus, beter bekent als een vierkant 8](#_Toc123997081)

[Figuur 13 3D hyperkubus, beter bekent als een kubus 8](#_Toc123997082)

[Figuur 14 4D hyperkubus, beter bekent als een tesseract 8](#_Toc123997083)

[Figuur 15 2D simplex, ofwel een driehoek 8](#_Toc123997084)

[Figuur 16 3D simplex, ofwel een tetraëder 8](#_Toc123997085)

[Figuur 17 4D simplex, ofwel een pentachoron 8](#_Toc123997086)

[Figuur 18 Voorbeeld van een fractaal 10](#_Toc123997087)

[Figuur 19 Mandelbrot set 10](#_Toc123997088)

[Figuur 20 Driehoek van Sierpiński 10](#_Toc123997089)

[Figuur 21 Fractal noise met amplitude 20 10](#_Toc123997090)

[Figuur 22 Fractal noise met amplitude 100 10](#_Toc123997091)

[Figuur 23 Fractal noise met amplitude 300 10](#_Toc123997092)

[Figuur 24 Fractal noise met frequentie 1 11](#_Toc123997093)

[Figuur 25 Fractal noise met frequentie 0.5 11](#_Toc123997094)

[Figuur 26 Fractal noise met frequentie 2 11](#_Toc123997095)

[Figuur 27 Fractal noise met 1 octaaf 11](#_Toc123997096)

[Figuur 28 Fractal noise met 3 octaven 11](#_Toc123997097)

[Figuur 29 Fractal noise met 6 octaven 11](#_Toc123997098)

[Figuur 30 Fractal noise met 10 octaven 11](#_Toc123997099)

[Figuur 31 Fractal noise met lacunariteit 2 12](#_Toc123997100)

[Figuur 32 Fractal noise met lacunariteit 1 12](#_Toc123997101)

[Figuur 33 Fractal noise met lacunariteit 3 12](#_Toc123997102)

[Figuur 34 Fractal noise met persistentie 0,5 12](#_Toc123997103)

[Figuur 35 Fractal noise met persistentie 1 12](#_Toc123997104)

[Figuur 36 Fractal noise met persistentie 0,1 12](#_Toc123997105)

[Figuur 37 Wereld gemaakt met voxels 14](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997106)

[Figuur 38 Verschil tussen voxels en polygons 14](#_Toc123997107)

[Figuur 39 Polygon met alle onderdelen aangeduid 14](#_Toc123997108)

[Figuur 40 Standaardobjecten van Unity 14](file:///H:\Jonathan\School\3de%20hogeschool\POP\2223pop-jonathanvercammen\Verslag%20PGT.docx#_Toc123997109)

[Figuur 41 Performantie bij ~5000 blokken 14](#_Toc123997110)

[Figuur 42 Performantie bij ~10 000 blokken 15](#_Toc123997111)

[Figuur 43 Performantie bij ~20 000 blokken 15](#_Toc123997112)

[Figuur 44 Vlak terrein 15](#_Toc123997113)

[Figuur 45 Sinusvormig terrein 16](#_Toc123997114)

[Figuur 46 Terrein met één laag noise 17](#_Toc123997115)

[Figuur 47 Terrein met vier lagen noise 17](#_Toc123997116)

# Bronvermelding

Amazon. (2022, May 4). *FastNoise Gradient*. Opgehaald van Amazon: https://web.archive.org/web/20220504014656/https://docs.aws.amazon.com/lumberyard/latest/userguide/component-gradients-fastnoise.html

Ambrosio, G. (2021). *Why Minecraft is a Technical Feat | Explaining the Engineering Behind an Indie Icon*. Opgehaald van Youtube: https://www.youtube.com/watch?v=CSa5O6knuwI

Archer, T. (2011). *Procedurally Generating Terrain.* Opgehaald van micsymposium: https://micsymposium.org/mics\_2011\_proceedings/mics2011\_submission\_30.pdf

Artjamayka. (sd). Opgehaald van Pinterest: https://www.pinterest.com/pin/515451119850762907/

Bycer, J. (2018, May 16). *A Study Into Replayability -- Random vs. Procedural Generation*. Opgehaald van Game Developer: https://www.gamedeveloper.com/design/a-study-into-replayability----random-vs-procedural-generation

Kelly, B. (2017, November 19). *How to render a fractal, fast*. Opgehaald van blog.bede: https://blog.bede.io/how-to-render-a-fractal-fast/

Kniberg, H. (2022, February). *Minecraft terrain generation in a nutshell*. Opgehaald van Youtube: https://www.youtube.com/watch?v=CSa5O6knuwI

Kwon, K.-R. (2019, October 20). *A Deep Learning Method for 3D Object Classification Using the Wave Kernel Signature and A Center Point of the 3D-Triangle Mesh*. Opgehaald van mdpi: https://www.mdpi.com/2079-9292/8/10/1196

Peters, K. (2021, July 17). *Perlin vs. Simplex*. Opgehaald van bit-101: https://www.bit-101.com/blog/2021/07/perlin-vs-simplex/

ScratchAPixel. (sd). *introduction polygon mesh*. Opgehaald van https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh.html

SimonDev. (2022, November). *The Absurd Usefulness of Noise in Game Development.* Opgehaald van Youtube: https://www.youtube.com/watch?v=sChQCdbLdHE

Wikipedia. (2022, September 13). Opgehaald van White noise: https://en.wikipedia.org/wiki/White\_noise

Wikipedia. (2022, Octobre 23). Opgehaald van Linear congruential generator: https://en.wikipedia.org/wiki/Linear\_congruential\_generator

Wikipedia. (2022, October 14). *Perlin Noise*. Opgehaald van Wikipedia: https://en.wikipedia.org/wiki/Perlin\_noise

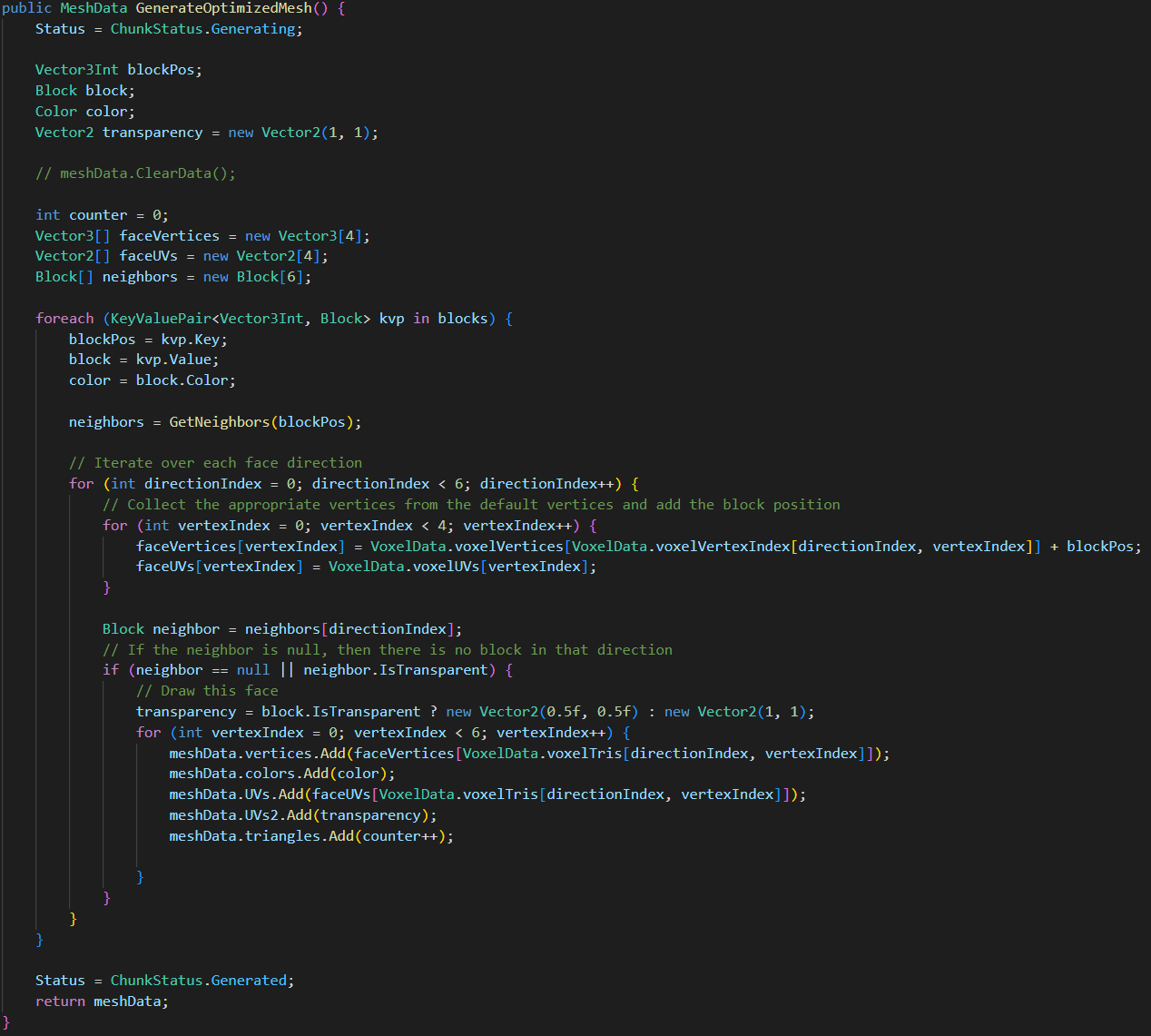
Wikipedia. (2022, December 16). *Sierpiński triangle*. Opgehaald van Wikipedia: https://en.wikipedia.org/wiki/Sierpi%C5%84ski\_triangle

Wikipedia. (sd). *Simplex*. Opgehaald van Wikipedia: https://en.wikipedia.org/wiki/Simplex

Wikitionary. (2022). Opgehaald van https://en.wiktionary.org/wiki

Yizzam. (2018, July 11). *The amazing world of fractals*. Opgehaald van blog.yizzam: https://blog.yizzam.com/the-amazing-world-of-fractals/

# Bijlagen



Codefragment 7 GenerateOptimizedMesh methode

[[1]](#footnote-1)

1. Verslag template van Richtpunt Campus Hamme [↑](#footnote-ref-1)