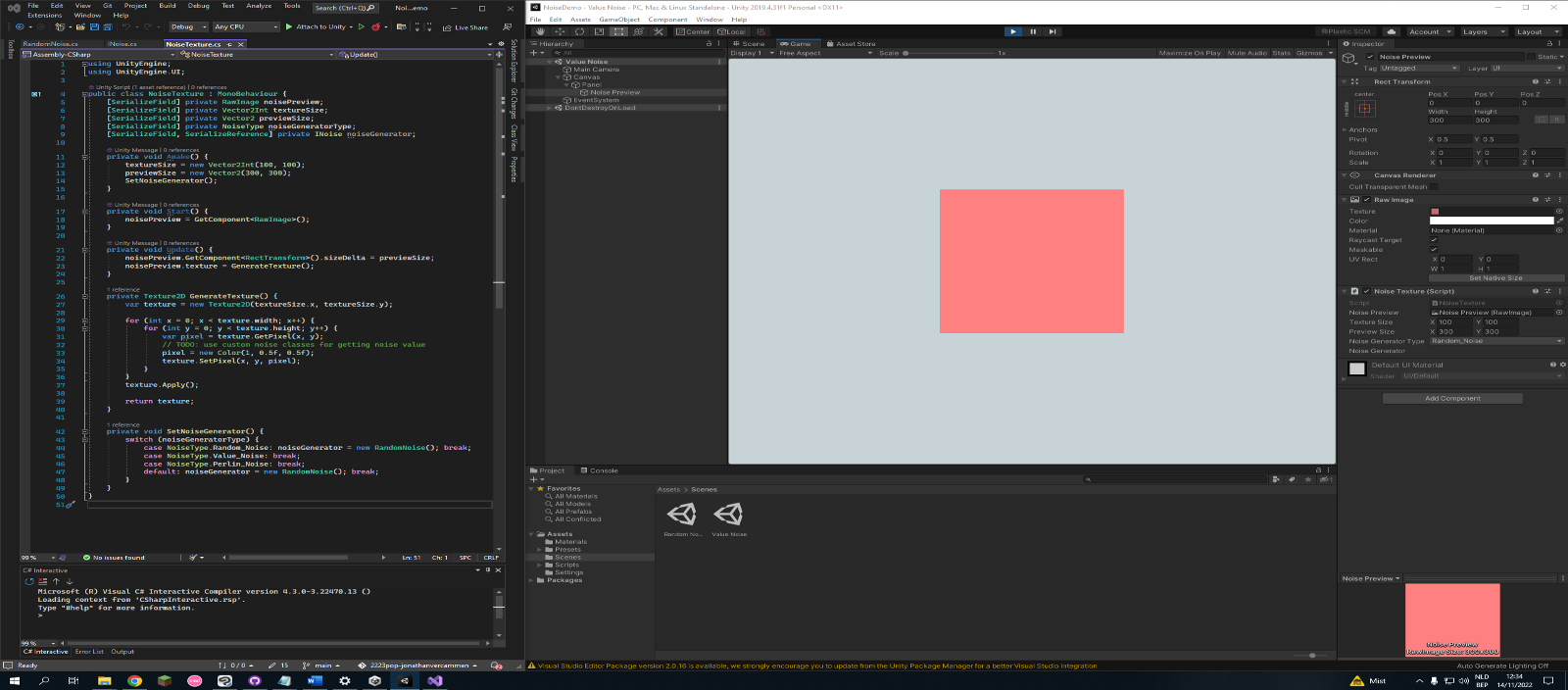
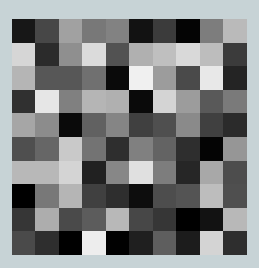
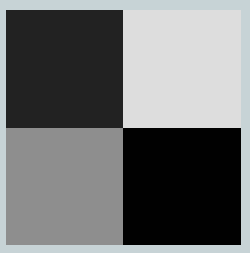
**14/11/2022:**

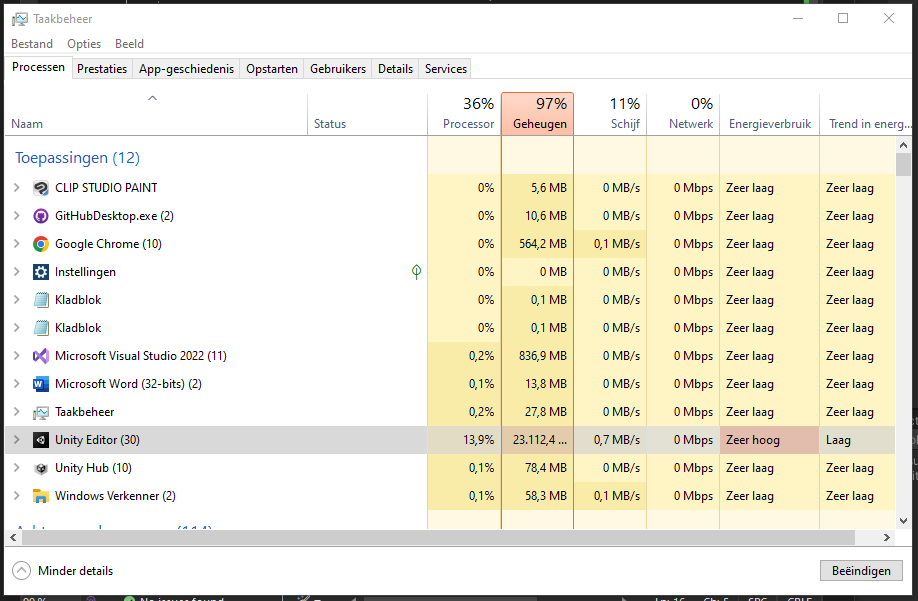
Ik ben begonnen met een script te schrijven om een texture in Unity aan te passen in code. Voorlopig geef ik alleen een rode kleur aan deze texture.



Na een kleine aanpassing kan ik random grijswaardes toewijzen aan de texture. Dit gaf niet echt het resultaat dat ik verwachtte. Het leek meer op value noise dan op random noise door de interpolatie. Na even zoeken zag ik dat een texture een filtermode heeft. Door deze op Point te zetten kreeg ik wat ik wou: random noise. 

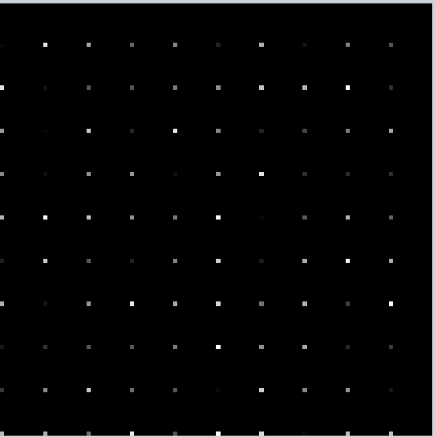
   

Tip van de dag: speel nooit met de tijdsinstellingen van Unity...

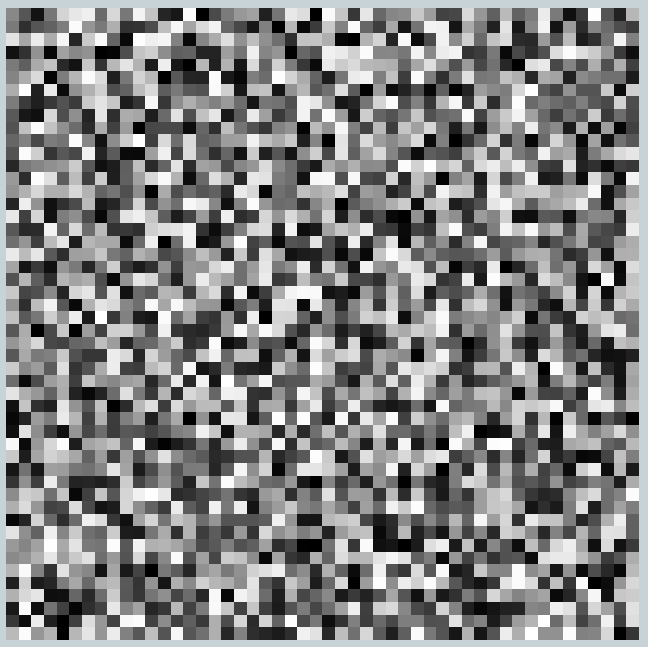
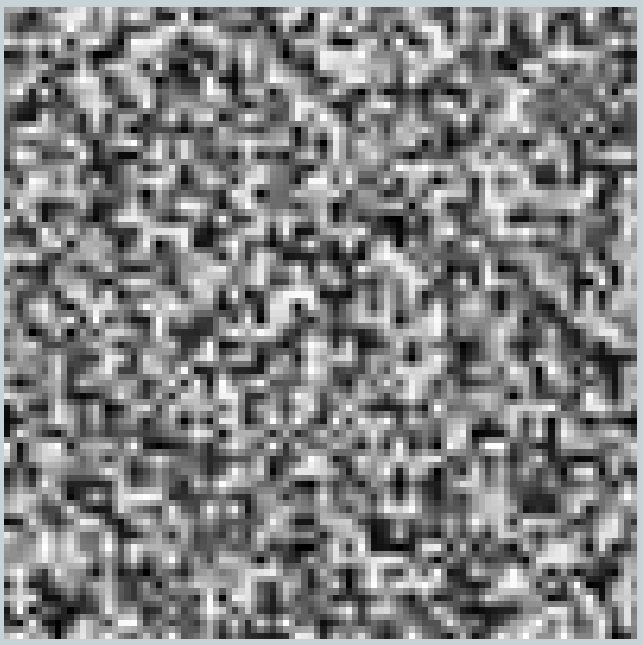


**15/11/2022:**

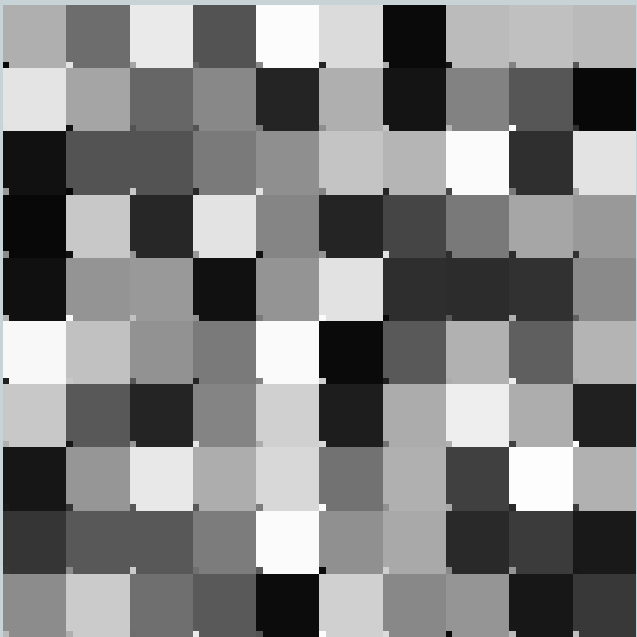
Vandaag begin ik aan het script voor value noise. Normaal zou dit niet al te moeilijk moeten zijn. Eerste poging:

 Het enige dat hier ontbreekt is interpolatie tussen alle punten

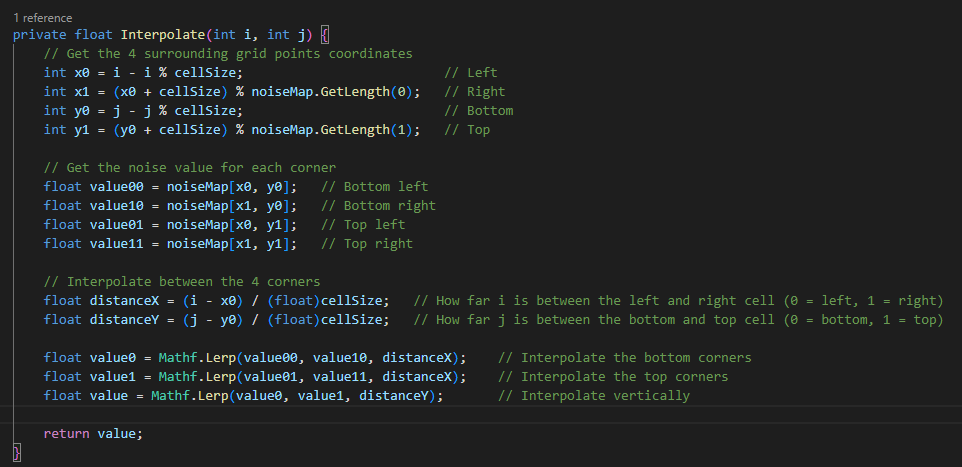
Eerste interpolatie werkte niet goed. Blijkbaar moet je bij deling tussen twee integers zelf de uitkomst casten naar een float.

Een paar leuke effecten:



Tip van de dag: gebruik comments. Ik weet niet of ik dit blok code nog zou begrijpen na een week.



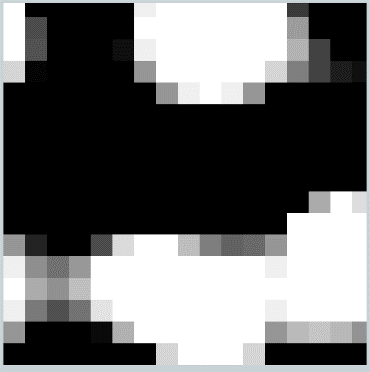
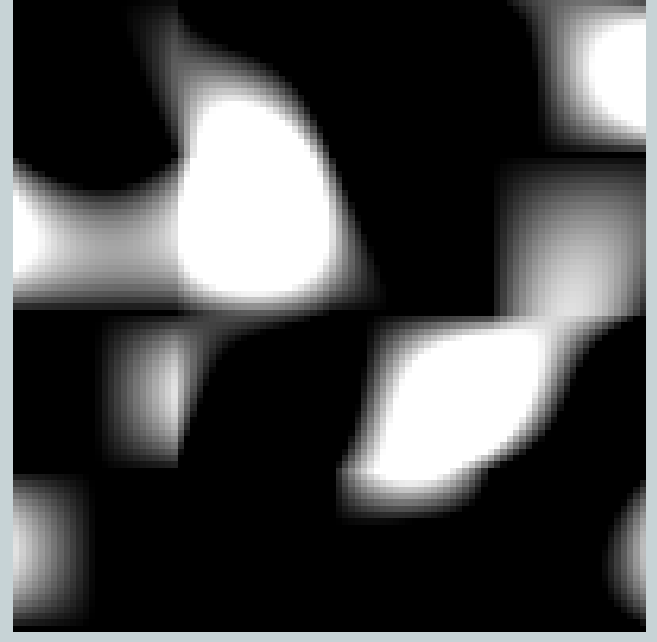
**17/11/2022:**

Perlin noise verloopt moeilijker dan verwacht. Als ik hier te lang mee sukkel ga ik waarschijnlijk de ingebouwde Unity-methode gebruiken.



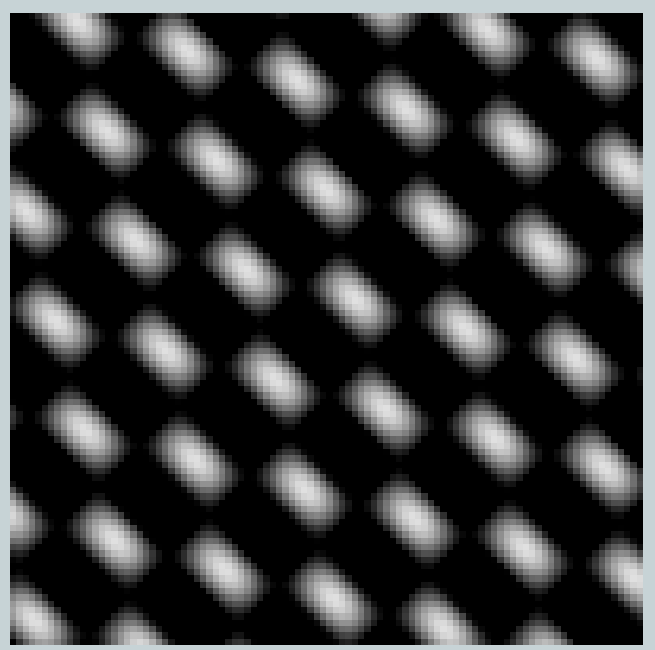
**18/11/2022:**

Ik kom stillaan dichterbij echte Perlin noise maar dit gaat ver boven mijn hoofd. Ik verwacht niet veel goeds voor simplex noise..

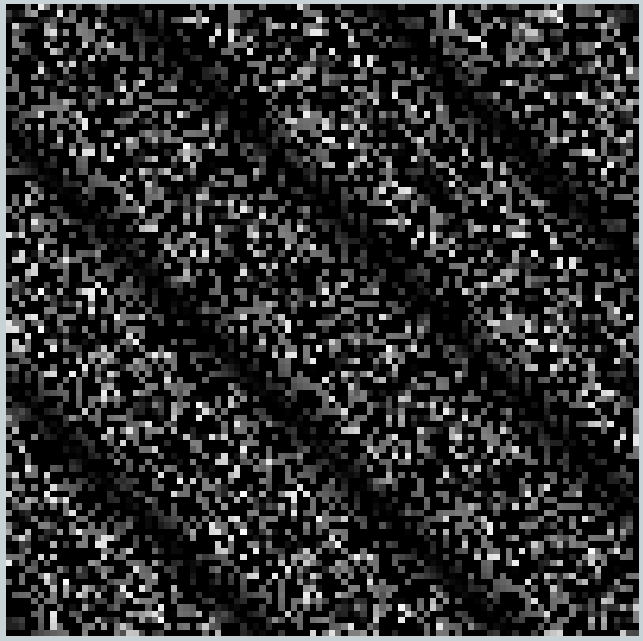
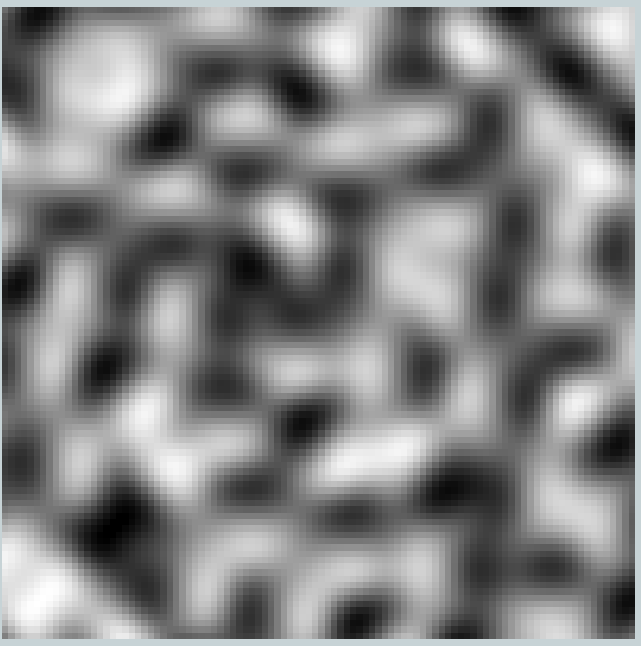
 

**19/11/2022:**

Simplex noise??

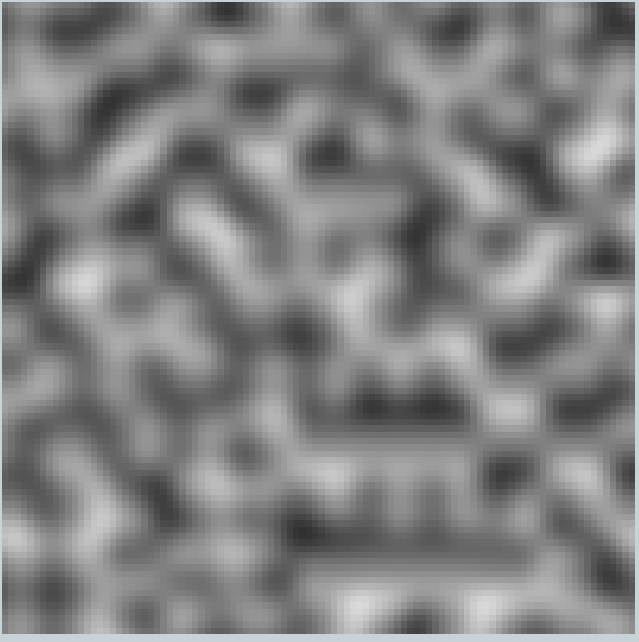
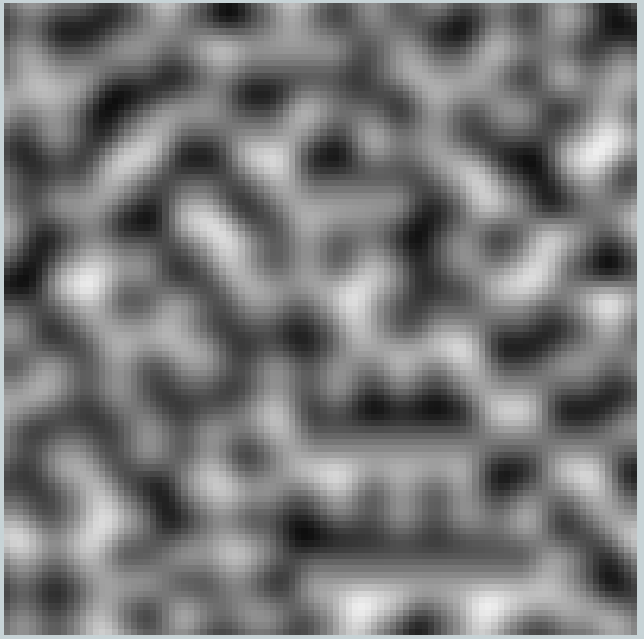
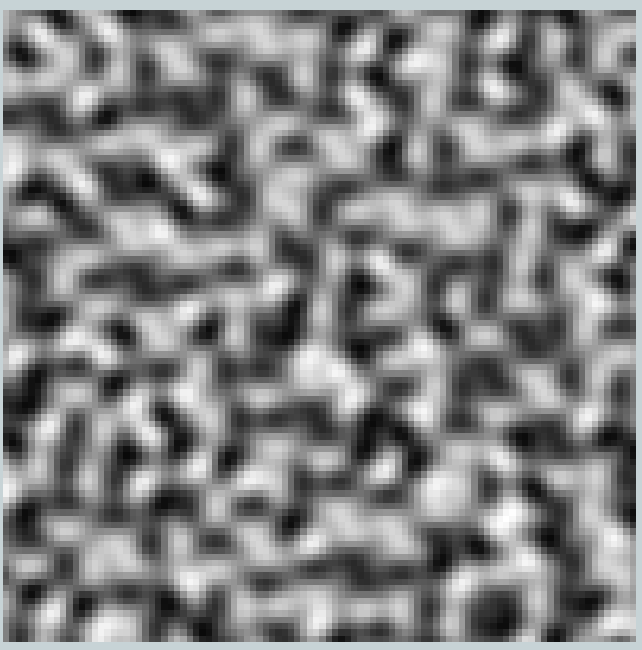


**20/11/2022:**

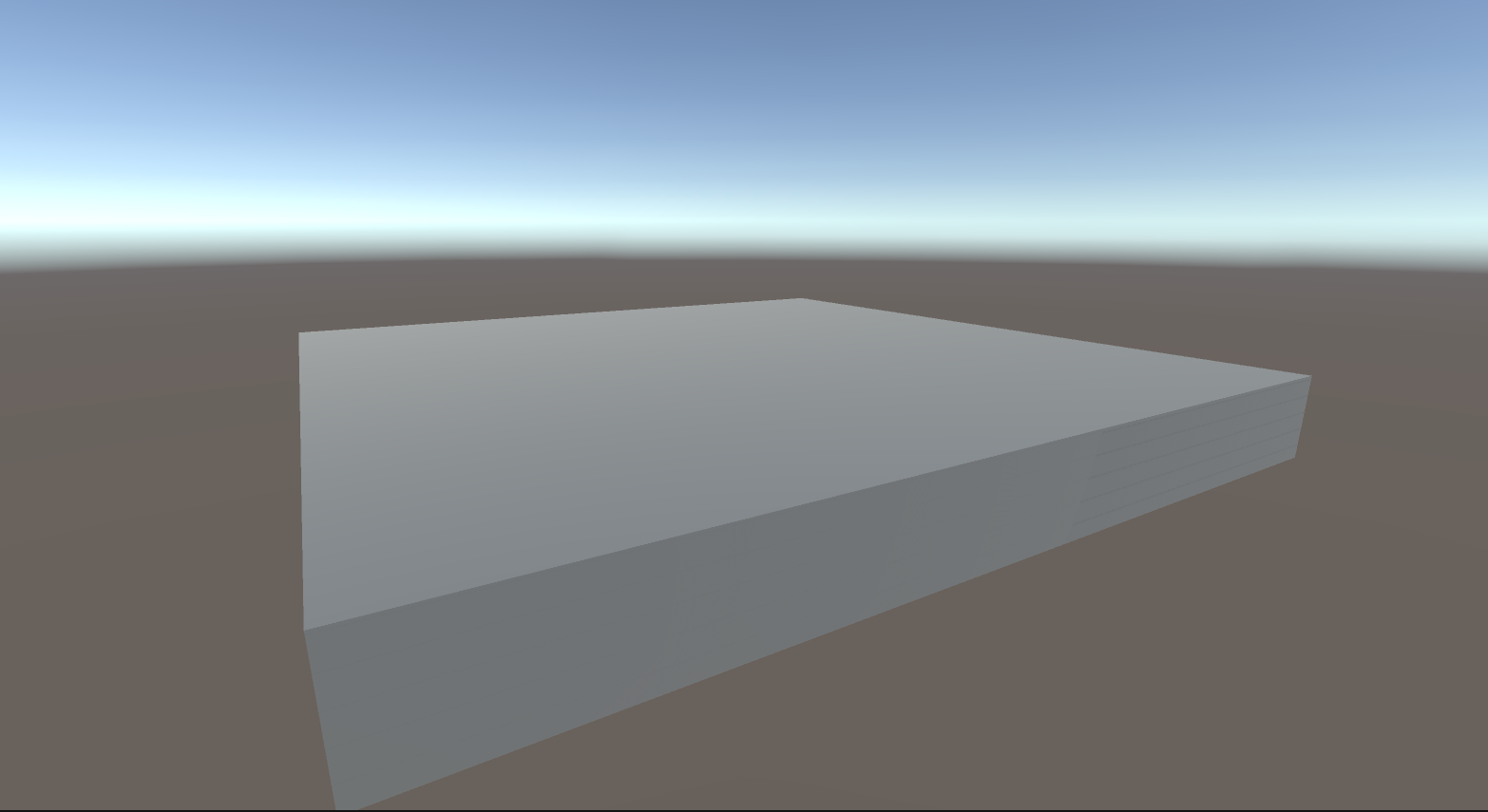
 

Na veel debuggen heb ik eindelijk gezien dat de scale van Perlin en simplex noise zeer klein is (~0.005) en niet de 1 – 100 range die ik gebruik in mijn noise texture. Bijkomend geven al deze functies een waarde tussen -1 en 1 terwijl ik een waarde tussen 0 en 1 wil voor grijswaardes. Later bij PGT kan ik wel de negatieve waardes gebruiken.

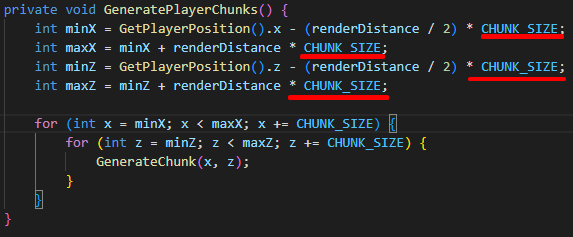
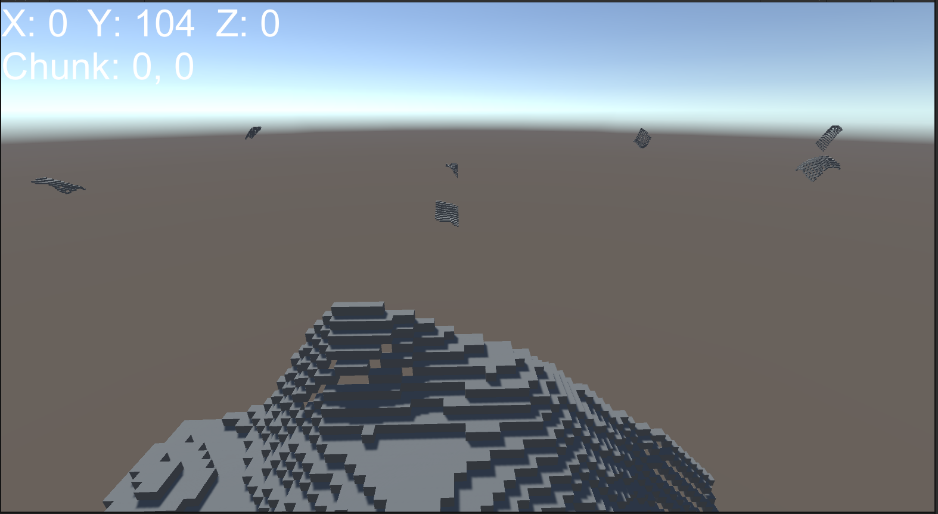
Custom Perlin noise Default Perlin noise Simplex noise

**27/11/2022:**

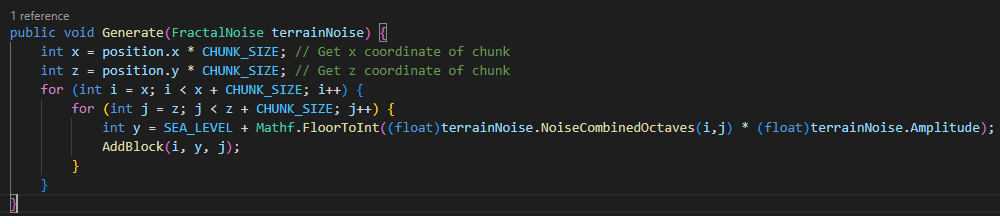
Eerste eenvoudige gegenereerd terrein. Er moet nog redelijk wat geoptimaliseerd worden want de standaard cubes van Unity zijn vrij intensief en niet super aanpasbaar. 

Eerste implementatie met Simplex noise

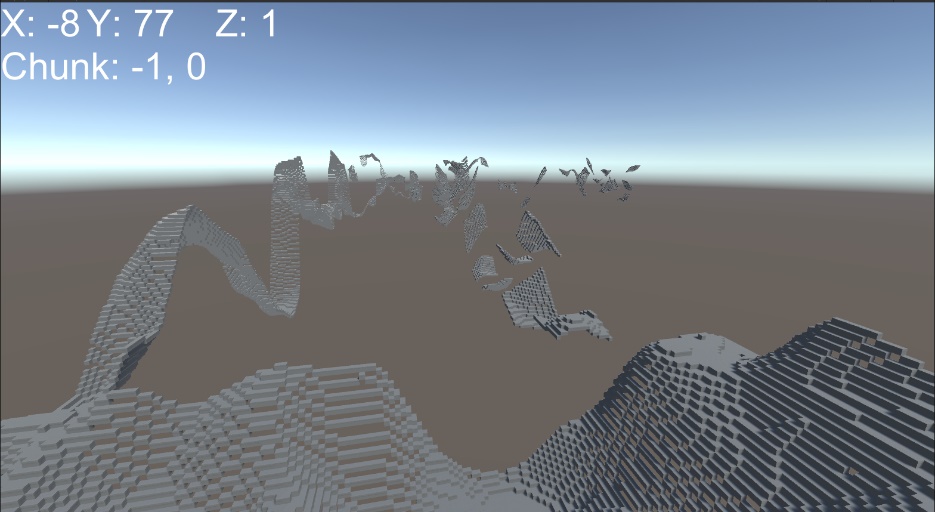


Terrein hangt niet mooi samen doordat ik alle coördinaten vermenigvuldigde met CHUNK\_SIZE. Een bijkomend probleem is dat ik deze waarde ook elke keer toevoeg aan mijn Chunk coördinaten (x & z). Dit zorgt ervoor dat ik chunks render die veel verder zijn want de GenerateChunk methode neemt al intern rekening met zijn grootte. Anders zou ik bijvoorbeeld de chunks (x) 0, 16, 32... renderen en niet 0,1,2...

Chunk Generate mtehode:



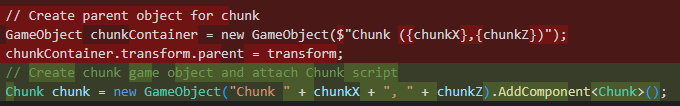
Ik zie nu ook dat ik nog een oplossing moet zoeken voor de parameter van deze methode. Als ik later meerdere noisemaps gebruik voor mijn terrein dan ga ik niet een parameter toevoegen per noisefunctie.

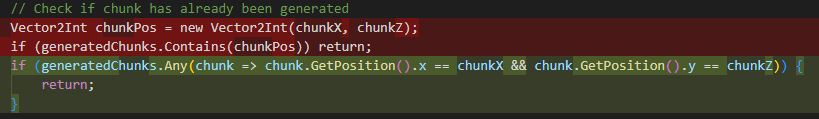


**30/11/2022:**

Vandaag werk ik vooral aan de optimalisatie van de code en zorgen dat er structuur in zit. Nu is het de bedoeling dat de terrainGenerator chunks genereert en elke chunk voor zichzelf alle blokken bijhoudt. Enkele optimalisaties zijn:

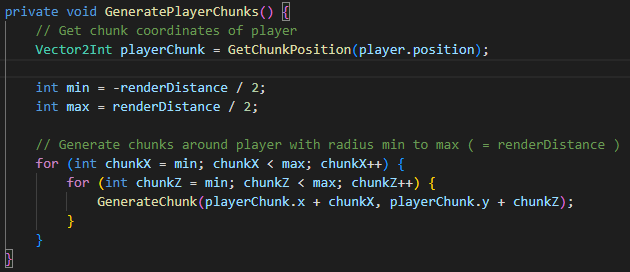
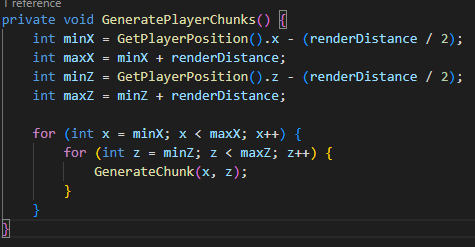
Chunk is nu zijn eigen klasse en niet gewoon een leeg GameObject.



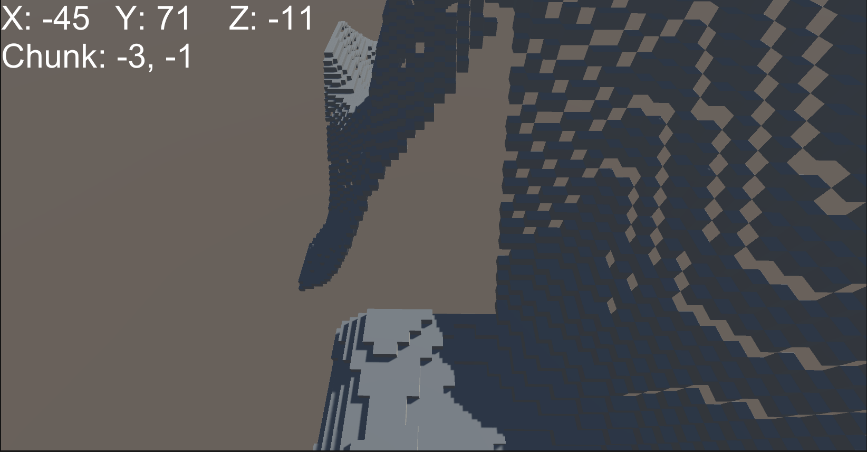
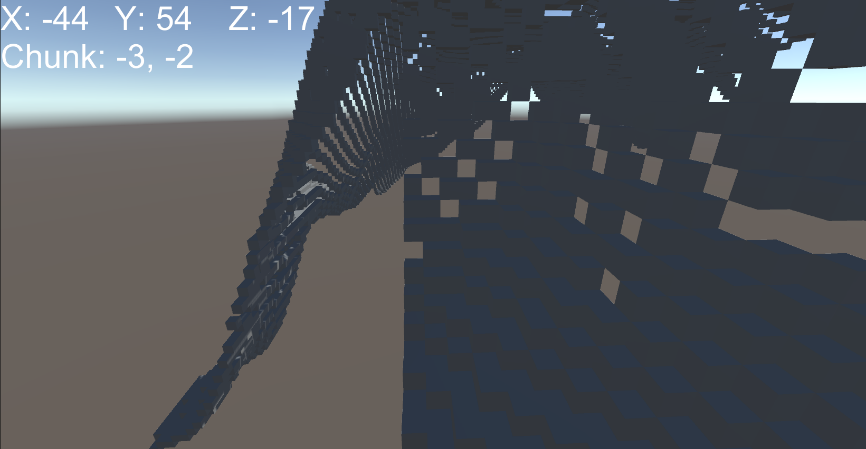
Daardoor heb ik een paar aanpassingen moeten maken zoals bij het kijken of een chunk al gegenereerd is. 

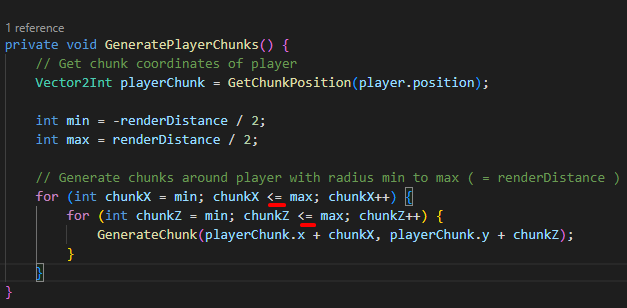
Nu alleen nog hetzelfde doen voor een blok.

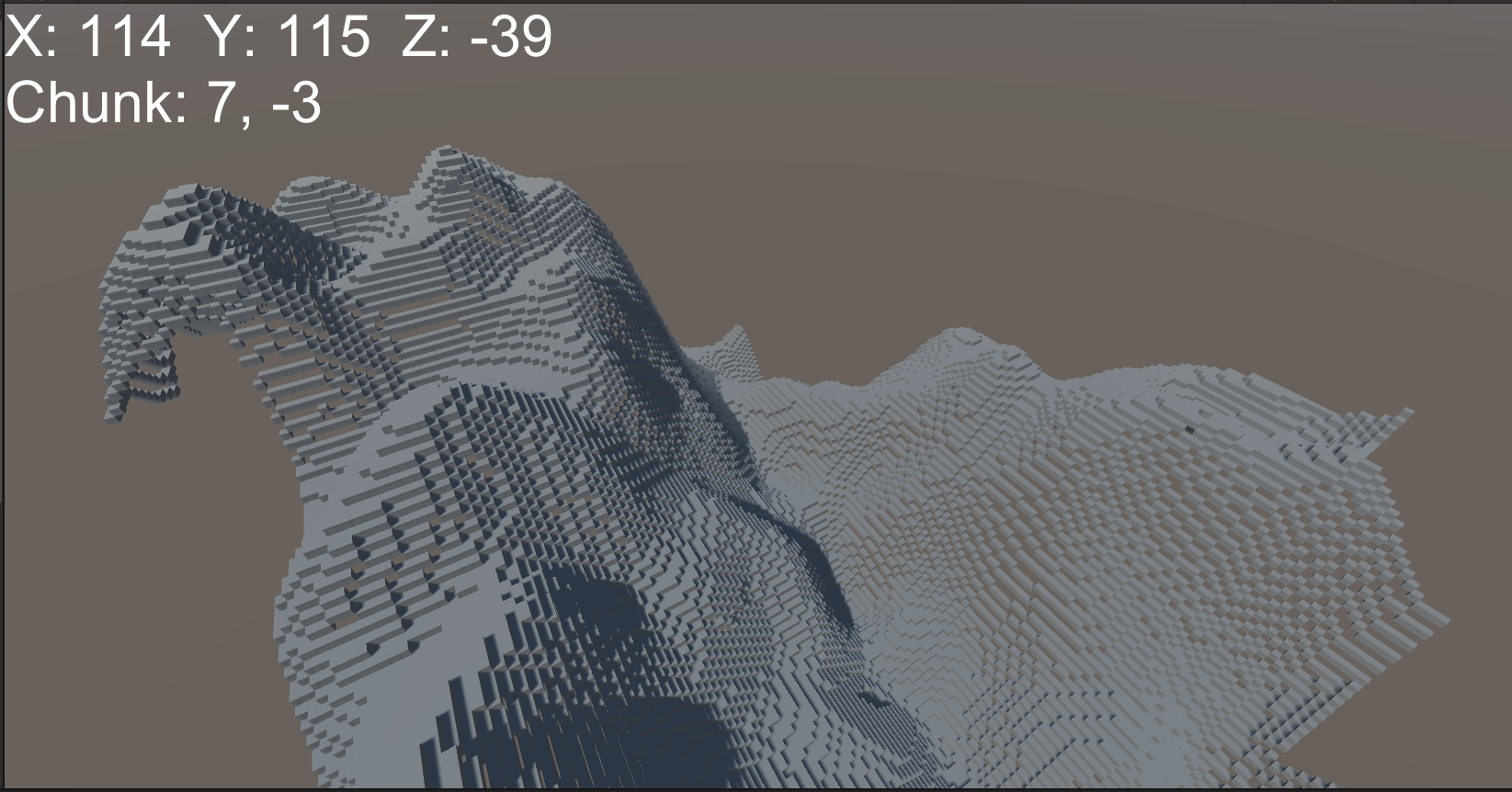
Ondertussen denk ik wat er mis loopt bij mijn chunk generation. De chunks worden te snel gegeneert maar volgen wel altijd de richting van de speler. Nu vraag ik direct de chunk op waarin de player zich bevindt en render ik alle chunks rond die chunk.



In X-richting werkt alles goed maar de Z-richting doet nog wat moeilijk. Deze chunks worden enkel gegenereerd wanneer ik in de chunk zelf sta.



De oplossing was vrij eenvoudig...



**3/12/2022:**

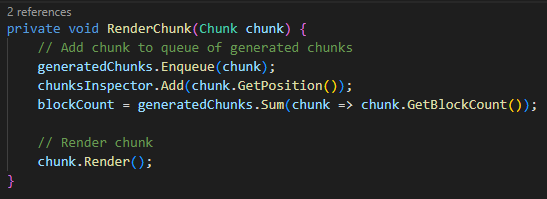
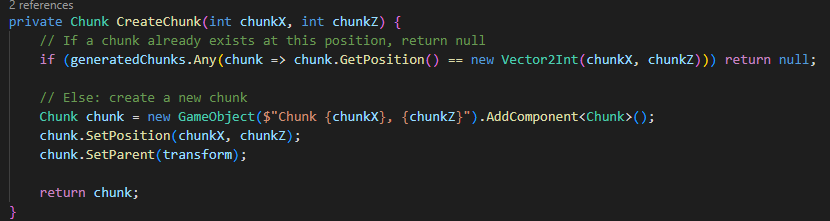
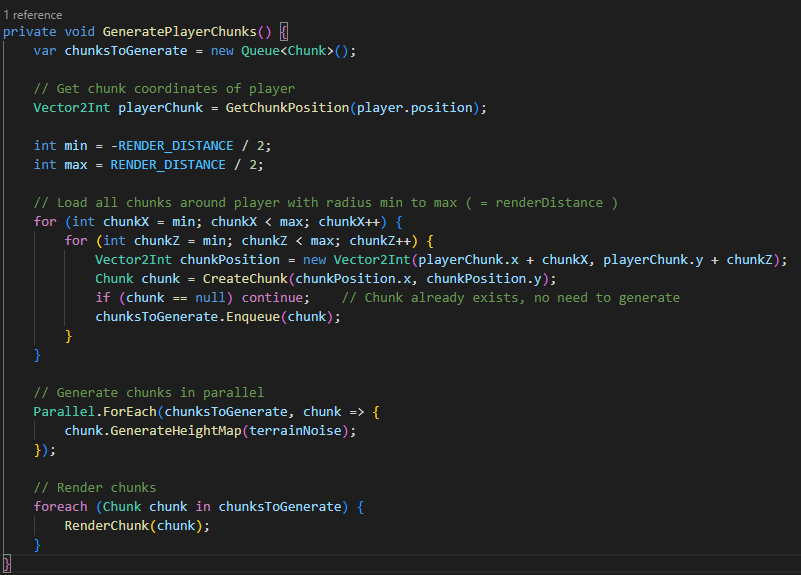
Er zijn al redelijk wat optimalisaties gebeurt ondertussen, vooral door parallel loops te gebruiken. Het grootste probleem blijft het renderen en instantiëren van GameObjects in Unity.

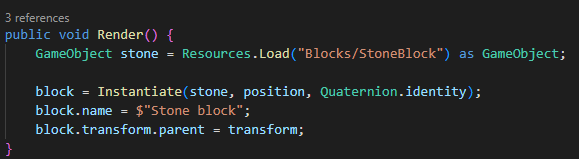
Hiervoor ga ik werken met meshes. In plaats van een chunk direct blok per blok te renderen, is het beter om eerst alle posities in de chunk te controleren.

Voor elk blok kijk je dan of er blokken naast staan, zo ja dan hoeft de zijde niet gerenderd te worden. Met al deze informatie zou ik dan een mesh kunnen genereren. Dit is veel efficiënter dan duizenden GameObjects te plaatsen maar vergt wel een beetje meer codeer werk. Ik ben niet van plan om het wiel opnieuw uit te vinden dus ga ik op zoek gaan op het internet naar delen code die ik kan gebruiken.

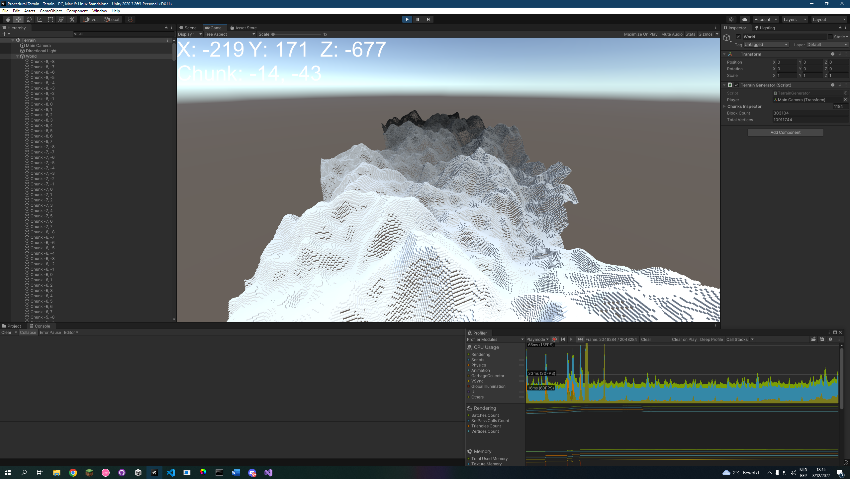
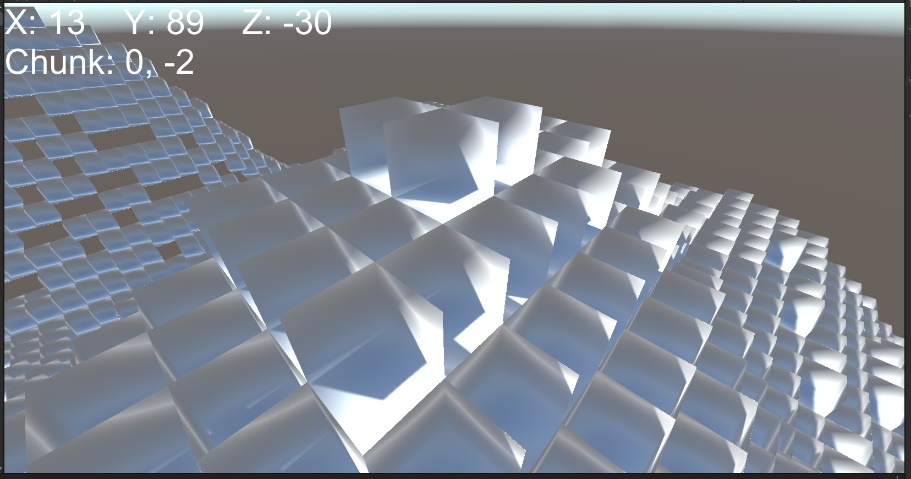
Uiteindelijk is het de bedoeling dat een Chunk al de posities van de blokken kent. Daarmee berekent hij de mesh en die mesh zal dan uiteindelijk gerendered worden op het scherm.

De code van chunk generatie ziet er nu nog zo uit.



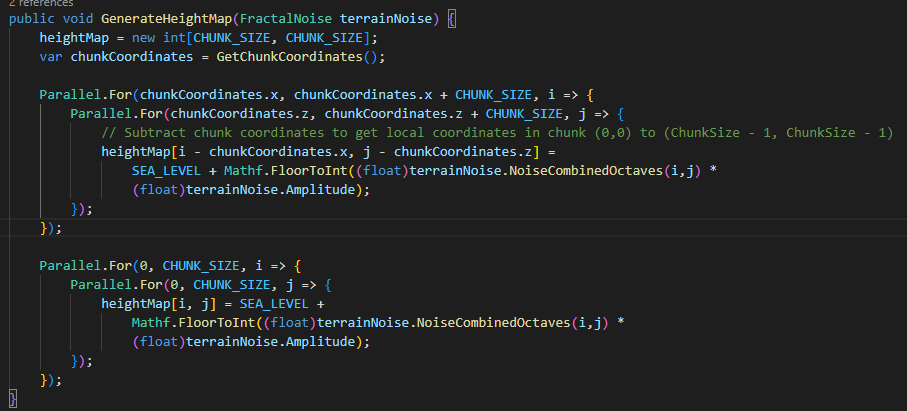
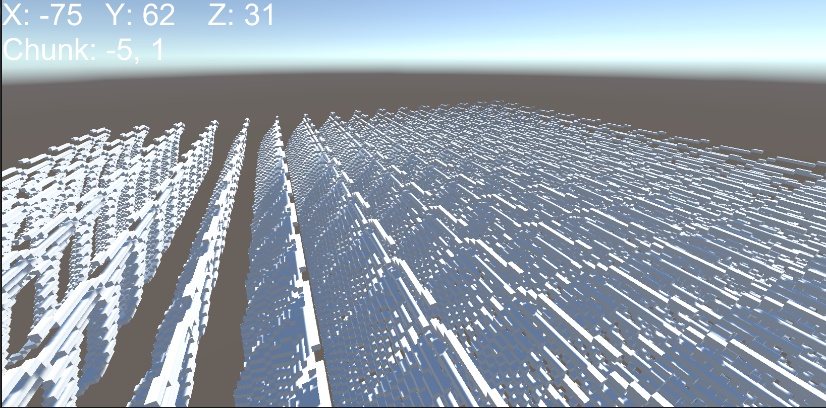


Er is een mesh, alleen nog niet wat ik wil.

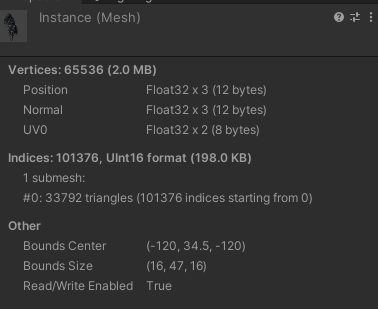
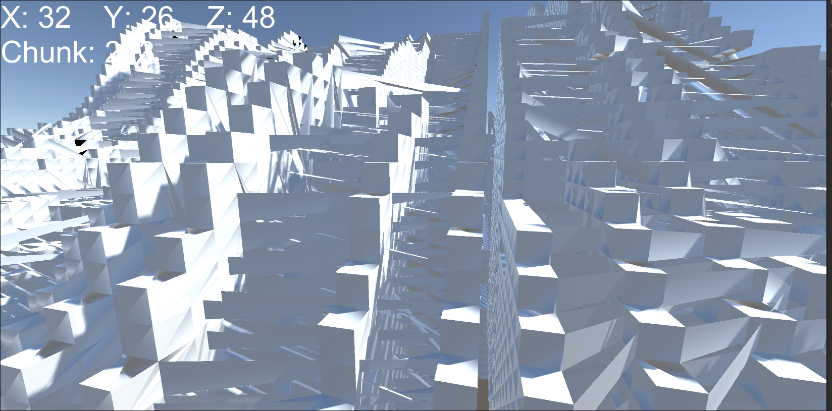


**4/12/2022:**

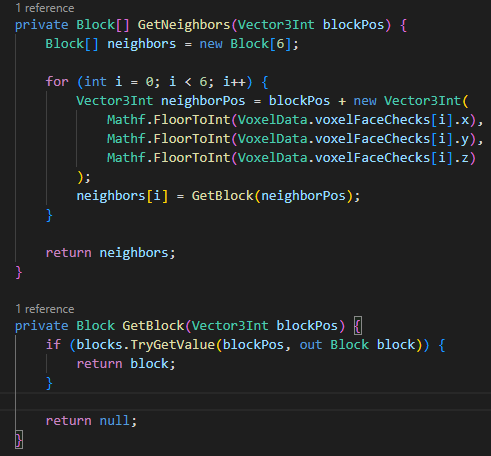
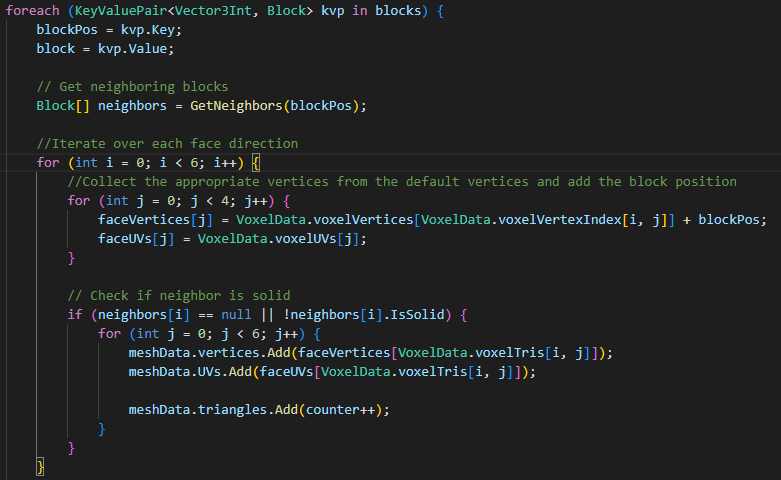
Ik dacht bijna een optimalisatie te vinden maar ik was vergeten dat (i,j) nodig is bij het berekenen van de noisevalue. Ze mogen dus niet altijd vanaf 0 vertrekken want dan krijg ik continu dezelfde waardes.



Mesh generatie is weer stuk. Unity heeft blijkbaar een limiet op de grootte van de mesh dus met andere woorden: meer mesh optimalisatie. Nu moet ik kijken naar de omliggende cubes om te bepalen of een zijde moet gerenderd worden.

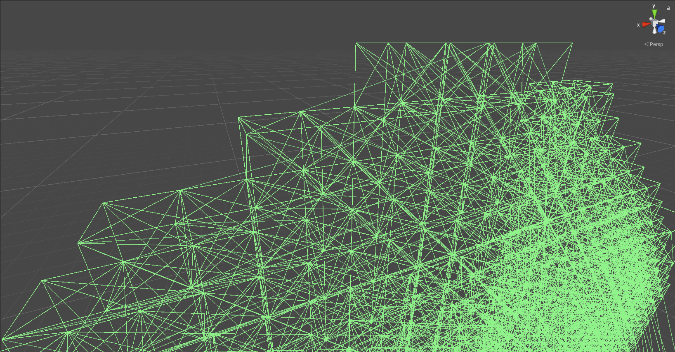
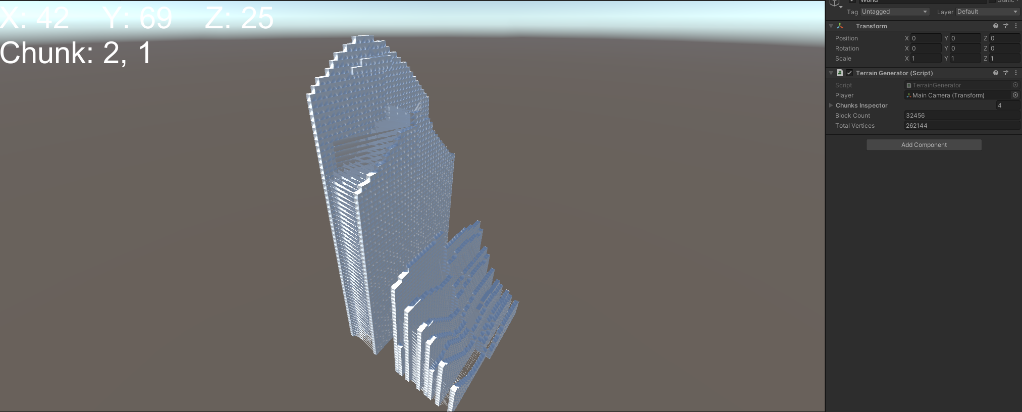


De mesh optimalisatie verliep vrij vlot deze keer. Nu kijk ik bij elke blok naar zijn buren voor ik de zijdes render.

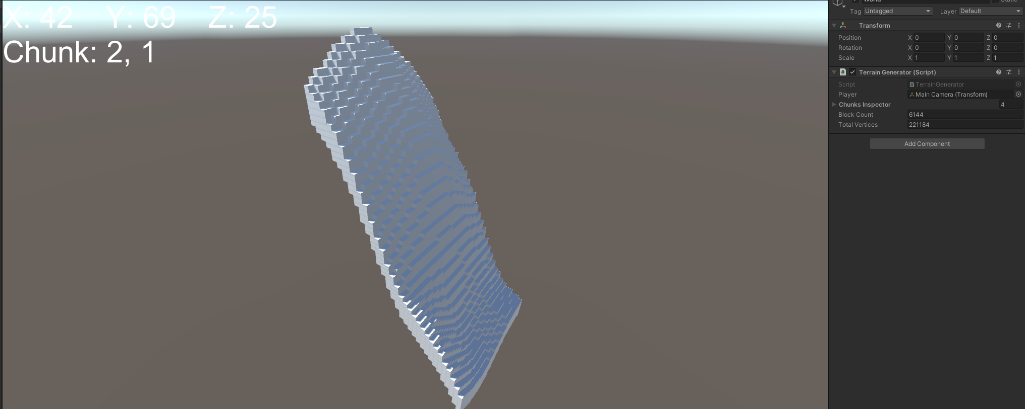


Dit helpt enorm voor performantie.

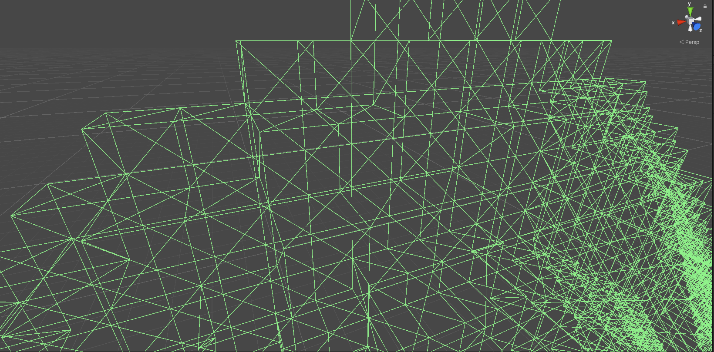
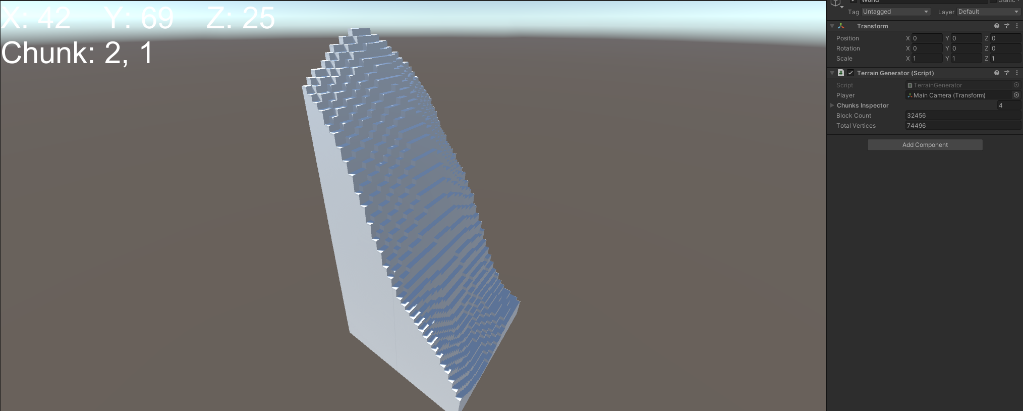
Zonder optimalisatie: 32 456 blokken, 262 144 vertices 🡺 Unity buffer overflow



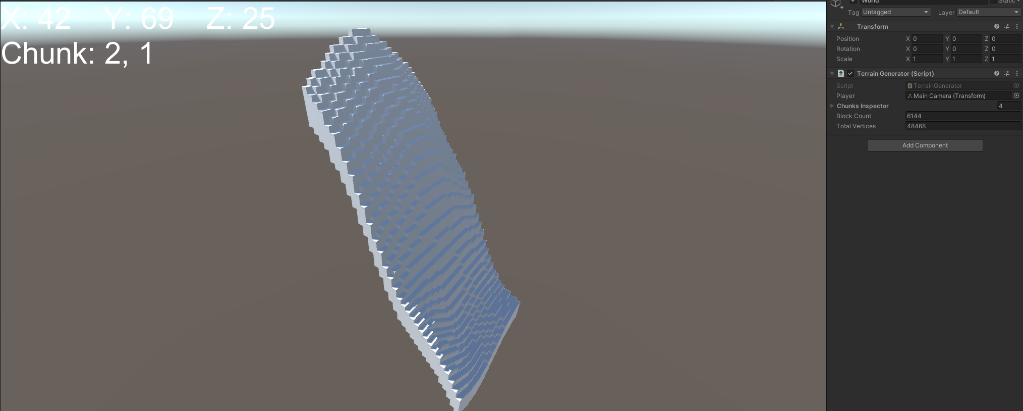
6 144 blokken, 221 184 vertices



Na optimalisatie: 32 456 blokken, 74 496 vertices (Vermindering van 187 648, ofwel 71,58%)



6 144 blokken, 48 468 vertices (Vermindering van 172 716, ofwel 78%)



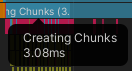
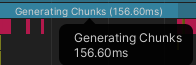
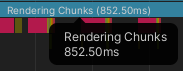
Op dit moment worden ook alle faces gerendered tussen chunks maar dit is niet zo’n groot probleem.

Nu is de grootste last van de GPU weg. Het aantal vertices is sterk verminderd.

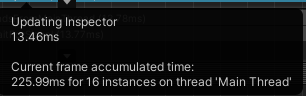
Alleen blijft de game altijd hangen tijdens het laden van nieuwe chunks.  Ik hoor de threads en tasks al komen.

1 seconde per frame om chunks te laden is redelijk lang. Voorlopig gebeurt alles in de main thread van Unity. Dus vanaf er een zware bewerking is, blijft Unity hierop vastzitten. De oplossing zou eenvoudig moeten zijn: gebruik een aparte thread om chunks in te genereren.

Na een beetje prutsen in de profiler van Unity heb ik ontdekt wat juist het langste duurt bij het genereren van de chunks.

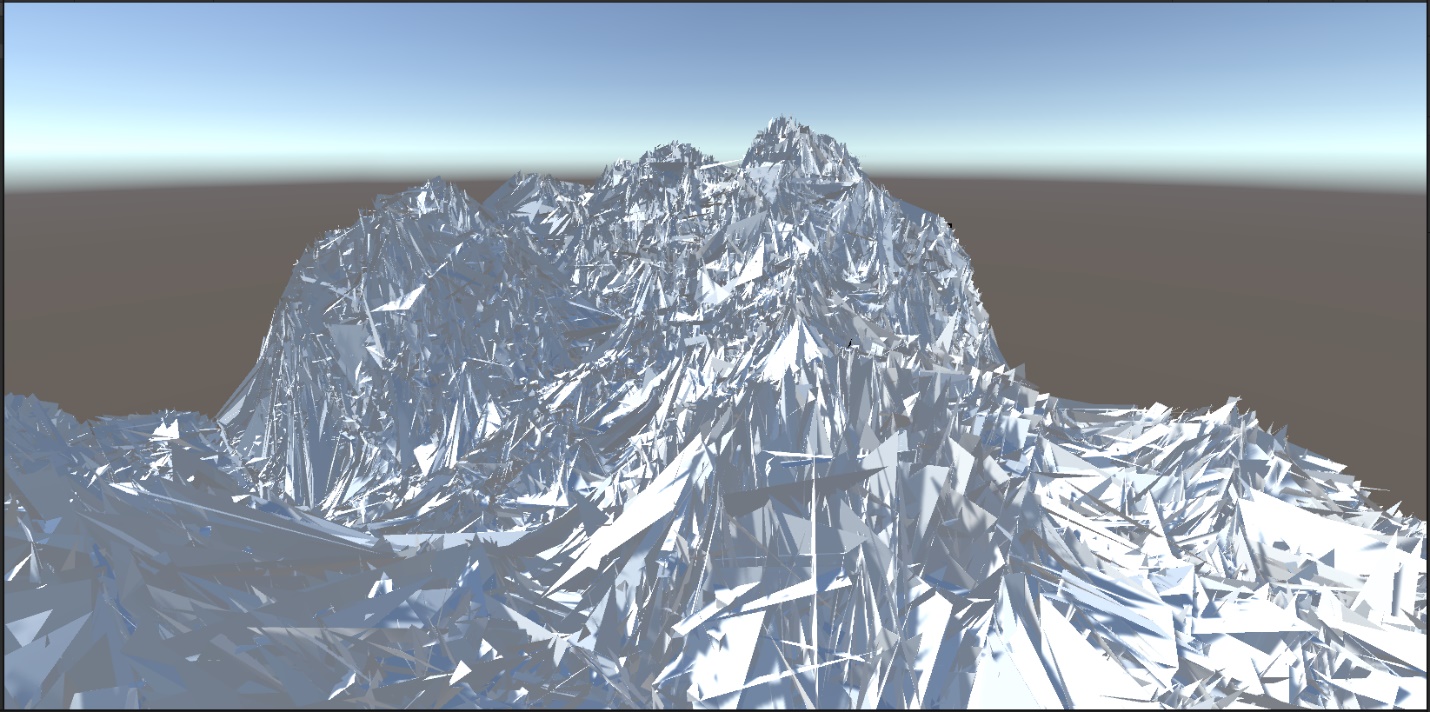
  

Ik ga beginnen met de grootste vertraging, het renderen. Na deze methode op te splitsen zie ik dat het updaten van de waardes in de Unity inspector veel tijd vraagt.

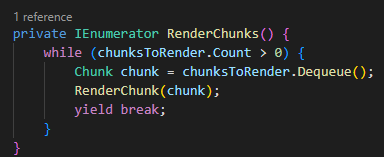
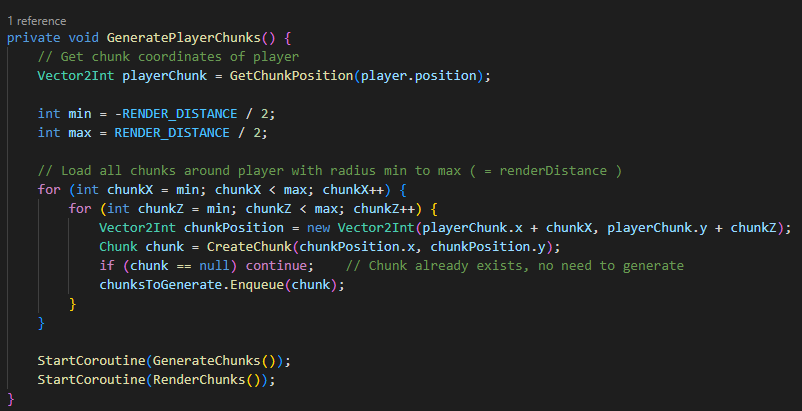
Ik neem aan dat als ik alle “debug calls” verwijder dat de performantie beter zal zijn.

Geen multithreading bij mesh generation, noted…



Dan maar eerst proberen om de chunk generatie te optimaliseren. Unity heeft een aantal manieren om asynchroon te werken. De default async – await, Tasks / Threads ofwel Coroutines en Jobs. Voor ik begin met multithreading ga ik eerst op zoek gaan naar andere optimalisaties.

Op dit moment gebruik ik Coroutines om de main thread van Unity meer frames te geven om iets te berekenen. Zo blijft heel het programma niet hangen telkens wanneer de player nieuwe chunks inlaadt. Code ziet eruit als volgt:



Elke yield statement zegt gewoon tegen Unity dat dit proces even on hold mag worden gezet. Zo kan de main thread andere dingen doen zoals de camera bewegen en input registreren. Normaal werkt Unity met verschillende Update loops. Hier ga ik later eens naar kijken.

Om alle chunks mooi bij te houden, gebruik ik vier queues. Waarschijnlijk kan dit efficiënter bijvoorbeeld met een dictionary<ChunkStatus, Chunk>() waarbij elke chunk zijn status zelf bijhoudt zoals: Initialized, Generating, Generated, Rendering, Rendered, Unloaded, Deleted...

