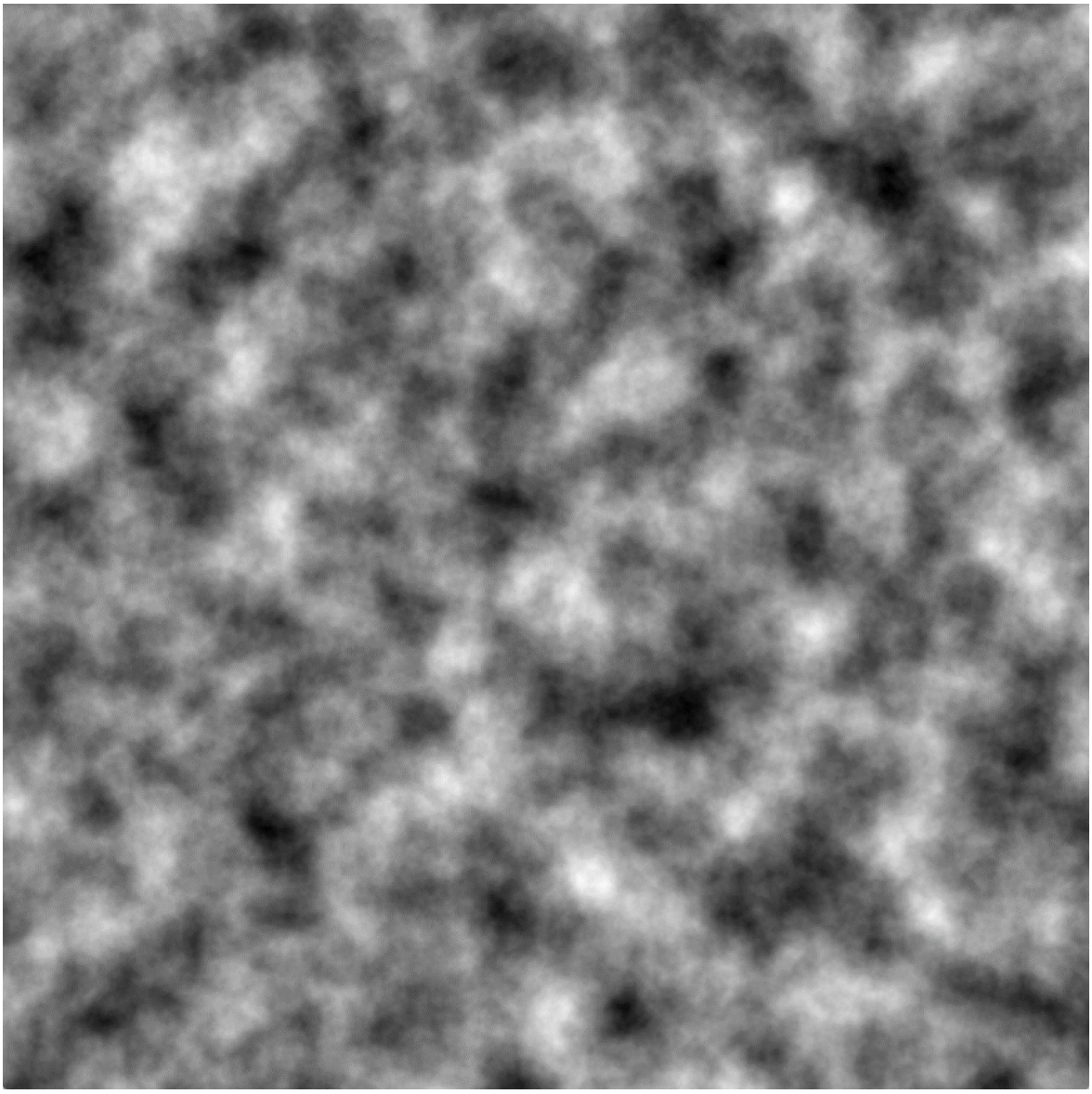
Procedural Verdensgeneration i spil



Fag: Digital Design og Udvikling/Teknikfag A, Programmering B

Skrevet i perioden 11-12-2023 til 19-12-2023

Skrevet af Andreas Nissen Riedel

Klasse: 3.i

Vejledere: Maya Saietz, Jakob Scheunemann Hastrup

H.C Ørsted Gymnasiet, Frederiksberg

Opgaveformulering:

Over de seneste årtier, efterhånden som teknologien har udviklet sig, er det blevet populært at lave open world computerspil, hvor en spiller frit kan udforske en virtuel verden i alle retninger. At skabe en sådan verden, som er både stor og interessant nok til at underholde spillere i mange timer, er et stort arbejde. I dette projekt skal du undersøge metoder til at generere sådanne verdener *proceduralt*, altså ved brug af en algoritme.

• Redegør for almindelige metoder til procedural generering af verdener til open world spil.
• Lav en komparativ analyse mellem to udvalgte spil: Et med en håndlavet verden, og et med en proceduralt genereret verden. Analysens fokus skal ligge på terrænets egenskaber.
• Byg en prototype af et spil med en proceduralt genereret verden. Redegør for de metoder du bruger. Inddrag relevante dokumentationsformer, f.eks. pseudokode eller flowdiagrammer.
• Diskuter og vurder prototypens kvalitet ifht. terrænets egenskaber.

Resumé

I denne opgave forklares ofte anvendte metoder til procedural generation primært i relation til terræn i spil. Målet er at finde ud af hvordan procedural generation kan bruges til at skabe open-world spilverdener. Det primære værktøj til at opnå dette er Perlin noise. Som værktøj kan det bruges til at skabe tilfældige talværdier, der har relation til hinanden. Det vil sige at der er et lag af blødhed til overgangen mellem tallene, og kan derfor bruges til at simulere realistisk terræn med bløde bjerge og have. I modsætningen til dette er der også håndlavede verdener, som også kan anvendes i et open-world miljø. Komparativt har de to metoder deres egne anvendelser i forskellige scenarier af spil. For at få en uddybende forståelse af procedural generation med Perlin noise udvikles et demo-program af en voxel-baseret terrænmodel i Unity. Ud fra denne model generes et terræn hvor man kan se anvendelsen af Perlin noise samt hvordan ændringen i forskellige matematiske værdier har en effekt på det genereret terræn. Denne demo (med mere videreudvikling) vil være en terrænmodel der vil kunne anvendes som en procedural open-world verden.
Når man skaber en verden som spiludvikler er det derved absolut nødvendigt at forstå hvornår de forskellige typer verdener kan anvendes i forskellige typer af spil, som udvikler skal man tage denne beslutning når man skaber sin verden, da valget medfører rigtig stor betydning for spillets endelige gameplay.

Indholdsfortegnelse

Indhold

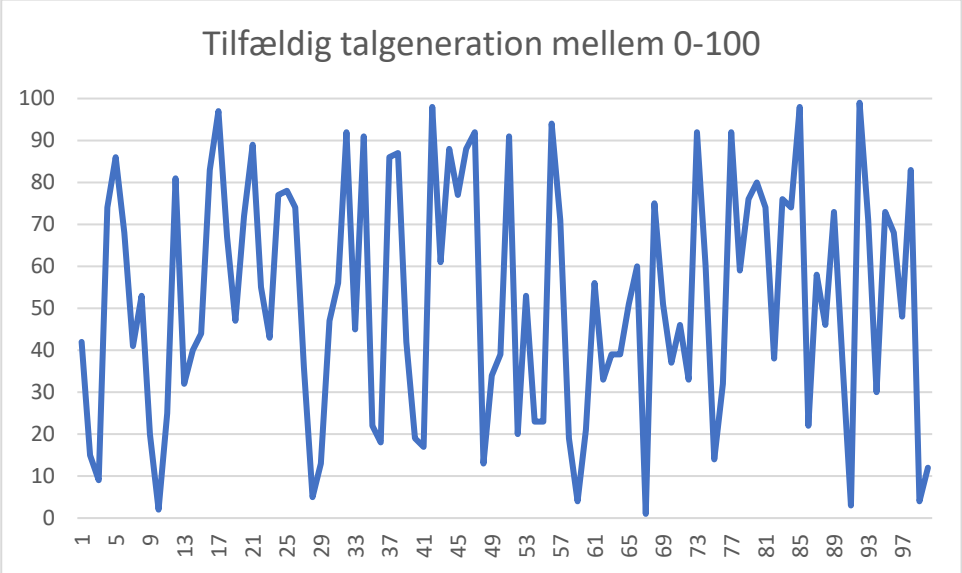
Indledning

At skabe terræn til spil kan ofte være en stor opgave. Procedural generation begynder derfor at blive brugt som en teknik der kan anvendes til at lave terræn i spil. Her vil jeg forklare almene teknikker til procedural generation og hvordan det ofte bruges i spil. Herunder vil jeg komparativt analysere de væsentligste forskelle og anvendelser af procedurale verdener og håndlavede verdener. Derudover udvikles en demo i Unity af hvordan Perlin noise kan anvendes til at lave terræn, og derved forklare hvordan det implementeres som kode. Denne terrænmodel anvender et voxelbaseret system til at lave en verden af blocks, herunder anvendes optimeringsteknikker som backface culling til disse voxels.

Almene Verdensgeneration teknikker

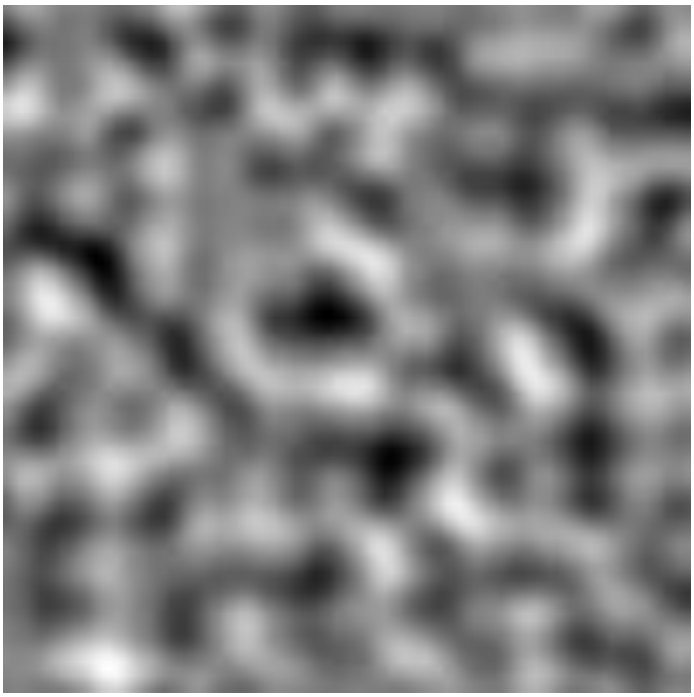
Perlin Noise

Et af de fundamentale værktøjer til procedural generation er tilfældighed. Det er det som gør at de ting, som bliver genereret, om det er teksturer, verdener, billeder osv., er unikke ved generation. Ren tilfældighed er dog upålideligt til dette, da der ikke er sammenhæng mellem det som bliver genereret. For eksempel, hvis vi kører en tilfældig talgenerator mellem 0-100, 100 gange, vil vi få en graf med ingen mærkbare mønstre i det. Hvilket ikke er særligt brugbart til realistisk generation af terræn, da vi ofte gerne vil have glidende overgange mellem tal.

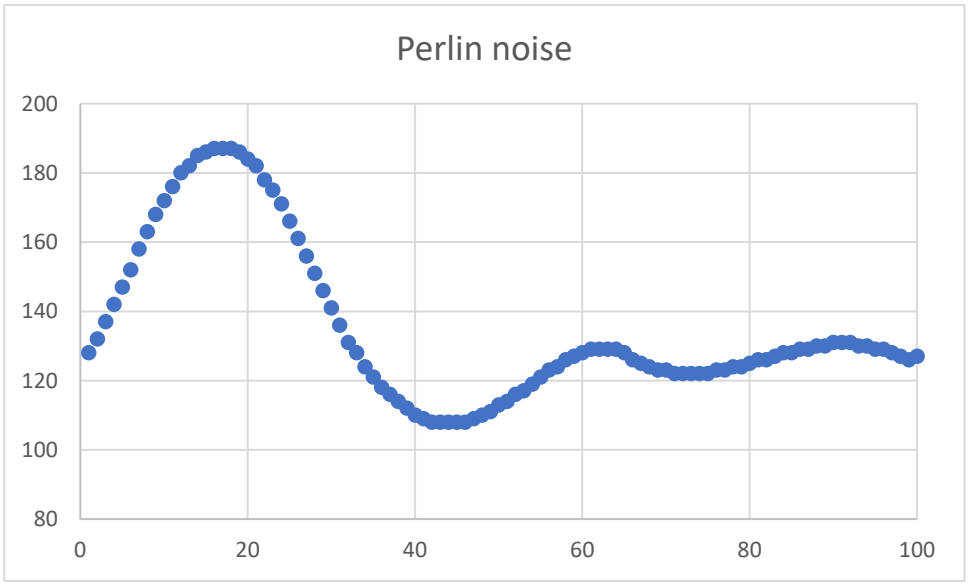


Figur 1 Graf over tilfældigt genererede tal mellem 0-100

Løsningen til dette problem blev udviklet af Ken Perlin i 1983 i form af Perlin noise. Tankegangen var at lave et automatisk genereret billede i sort og hvid hvor de forskellige farver kan repræsentere tilfældighed, f.eks kan helt hvid være 0, og helt sort være 100. Alle farver imellem de to, repræsenterer så alle tal mellem 0 og 100. Med denne tilgang sørger man for at der ikke er store forskelle mellem tallene, som var tæt på hinanden og der er derfor en glidende overgang i tilfældigheden bag det. Hvis vi sætter denne taldata fra Perlin noise ind som en graf, kan vi se at der stadig er tilfældighed bag det, men at der er et lag af interpolation mellem tallene, og derved er der talværdier som forbinder maksimummerne, og minimummerne.



Figur 2 Perlin noise, tilfældigt genereret ved brug af https://pypi.org/project/perlin-noise/



Figur 3 1D graf over et udsnit af Perlin noise

Perlin Noise med Python

Denne ovenstående graf blev lavet ud fra taldata fra Perlin noise. Denne taldata kan ekstraheres ved at bruge Pillow, numpy, perlin_noise og matplotlib til både at skabe Perlin noise, og få talværdier ud fra det.

Vi starter med at generere et PerlinNoise objekt, som vi så kan generere et billede ud fra ved at kigge på de tilfældigt genereret talværdier i objektet og lave et grayscale colormap ud fra dem.

```
6. noise = PerlinNoise(octaves=10, seed=1)
7.
8. xpix, ypix = 100, 100
9. pic = [[noise([i/xpix, j/ypix]) for j in range(xpix)] for i in range(ypix)]
10.
11. image_path = 'noise.png'
12. plt.imsave(image_path, pic, cmap="gray")
```

Efter dette, konverterer vi billedet til grayscale igen, for at simplificere overgangen mellem tallene. Uden dette, vil vi ende med floats med ekstremt mange kommatal, hvilket ikke er nødvendigt i vores mere generelle anvendelse her.

```
14. #Convert to grayscale for number simplifaction
15. img = Image.open(image_path).convert('L')
16. perlin_array = np.array(img)
```

Til sidst laver vi en liste hvor den første værdi af alle elementerne i vores array bliver tilføjet til. På den måde får vi alle tallene langs den første stribe af pixels på y-aksen af vores Perlin noise.

```
18. list = []
19. for x in perlin_array:
20.     list.append(x[0])
```

Til sidst gemmes denne taldata i en .txt fil, som vi så kan sætte ind i excel, eller et lignende program, og danne grafer med.

```
22. f = open("perlinarray.txt", "w+")
23. f.write(str(list))
24. f.close()
```

Hele dette program kan ses i bilag

Med dette værktøj er der blevet skabt en række af tilfældigt genereret billeder, teksturer og terrænmodeller. Et af de mest bemærkelsesværdige i relation til spildesign, er Minecraft. Dets terræn anvender 5 forskellige Perlin noises der alle styrer forskellige aspekter af terrænet, som f.eks. temperatur og elevation, dette uddybes yderligere i Udforskning

Håndlavede Verdener

Legend of Zelda: Breath of the Wild's verden

I et open-world, men stadig historiebaseret spil, er det svært at balancere friheden i udforske selv, og fremgang i historien. Dette var en af de sværere opgaver som Nintendo oplevede under produktionen af Legend of Zelda: Breath of the Wild (BotW). Deres mål med at skabe denne verden var at lave en oplevelse som var unik for hver spiller, hvor lige meget hvad man gjorde, og hvor man gik hen, at det på en eller anden måde var betydningsfuldt for spilleren, og at der ikke var nogen fast vej som spilleren var nødt til at følge. Dette gjorde at spilleren konstant skulle tage deres egen beslutninger, og derved forme deres egen måde at spille spillet, hvilket var præcis det som Nintendo gerne vil have at man oplevede spillet. For at opnå dette havde de nogle design principper som de fulgte, når det gjaldt for spillerens opfattelse af verden, og hvordan det havde en indflydelse på deres beslutninger undervejs.

Points of interest

For at opnå følelsen af fri udforskning lavede Nintendo en masse steder som spilleren naturligt vil gå hen imod. Dette var i form af Shrines, fjende baser, karakterer, som man kunne snakke med, Sheikah Towers og meget andet.

De primære historie-drivere er Sheikah Towers. De fungerer som store og let genkendelige tårne som der kan ses meget langt væk fra. At finde og skalere et betyder at alt omkringliggende terræn bliver synligt på dit mini-map, hvilket er værdifuld information for spilleren, og signalerer ofte

Figur 4 Sheikah Tower fra BotW, billede fra https://segmentnext.com/breath-of-the-wild-sheikah-towers/

fremskridt i historien. Disse tårne alene giver dog ikke en følelse af udforskning, da det ofte føles mere som en tråd man følger når man går fra tårn til tårn. Nintendo valgte derfor at lave en masse andre points of interest i form af shrines, hestestalde, monster camps, bjerge og masser andre ting. Ved at sprede disse steder rundt mellem tårnene gør det at spilleren kigger rundt på terrænet og løbende tager beslutninger om hvor de vil gå hen. Dette er ofte i form af en følelse hvor man inderligt siger: ”aha, hvad er det?”, og går derhen. Ved at sprede disse unikke steder rundt overalt i verden føles det som et langt brødkrummespor som spilleren følger, men som alligevel der forgrenet nok til at der ikke er krav til at du skal gøre det på en specifik måde.

Dette gør at lige meget hvor du går hen, vil der altid være et eller andet spændende, og derved en grund til at være der. Som spiller, gør det at du føler du har komplet kontrol over hvor du har valgt at gå hen og hvorfor. Som spildesigner er det perfekt, da spilleren har denne følelse, men alligevel ender de altid med at være de steder som driver historien videre.

Figur 5 Visualisering af forskellige ruter man kan tage mellem tårne, taget fra Game Maker’s Toolkit: https://youtu.be/CZzcVs8tNfE?si=wwAiMEz9TM9j6MQv&t=447 7:27 min

Trekants design

Hvis man kigger på verdenen af BotW, kan man se et mønster i hvordan terrænet er opsat. Næsten alting er en form for trekant. De rækker fra meget store i form af bjerge, helt ned til de mindste sten på højde med spilleren.

Figur 6 Forskellige størrelser af trekanter i BotW, fra https://www.blog.radiator.debacle.us/2017/10/open-world-level-design-spatial.html

Med denne tilgang til at lave terræn får man en række fordele. Man præsenterer spilleren for to beslutninger under udforskningen af verdenen. Klatrer man op ad bjerget? Eller går man rundt om? Begge beslutninger leder til at man finder noget nyt. Trekanterne gør også at man simpelthen kan gemme noget bag dem, fx et Sheikah Tower, og derved langsomt præsentere spilleren for noget nyt, som leder dem på et nyt brødkrummespor. Samtidigt fungerer de større trekanter som vartegn for spilleren, så det er nemmere at skabe overblik over ens position i verdenen. Når man kigger på en trekant, bliver ens øjne naturligt også ledt til toppen, man kan derfor, på større trekanter, vælge at ligge nogle points of interest på, eller inde i trekanten, som vi fx kan se i ”Dueling Peaks”.

Figur 7 Dueling Peaks, BotW

Deres opdelte design gør at der er plads inde i bjerget som man kan udforske, og selvom det er en trekant, kan man stadig se hvad der er bag dem, på grund af hullet der opdeler den i midten.

Komparativ analyse

Håndlavet og procedural genereret verdener er lavet til to vidt forskellige former for spil. Nogle gange kan procedural generation være brugt når man er et mindre spilfirma som har brug for en stor verden. I andre tilfælde er det fordi spillet helt essentielt er baseret på at verdenen er procedural, og at du frit kan udforske nyt terræn hver gang du spiller.

Om det er håndlavet eller procedural så har de hver deres fordele og ulemper som gør at de anvendes i specifikke scenarier alt efter hvad der passer bedst.

Storytelling

Hvis der er snakke om storytelling i et spil vil de fleste spil allerede der læne sig mere op ad håndlavet verdener. Se for eksempel på Red Dead Redemption 2, den gør sig brug af unikke lokationer og karakterer, der hver har deres eget sted i verdenen, og anvender derfor en hånd-

lavet verden. Overført til procedural, vil det betyde at de karakterer skulle være tilfældigt genereret, hvilke betyder at de ikke vil have nær så meget dybde. Selvfølgelig er det muligt at lave en håndlavet karakter i en ellers proceduralt verden se F.eks. Artemis fra No Man's Sky[1], men meget af deres dybde er væk, det betyder enten at de ikke fysisk er i verdenen, eller at der er en adskillelse mellem den procedurale og håndlavet del af spillet. Dette kunne være i formen af en "hub world", for eksempel, Space Anomaly i No Man's Sky[2]. Tag derimod et spil som Minecraft, og meget af dens gameplay går ud på at spilleren selv finder sine egne mål med spillet. Derfor er Minecraft et spil som gør rigtig god brug af en procedural verden, da der ikke er noget overordnet mål, eller en specifik måde at spille spillet, er der heller ikke nogen krav til at verdenen skal have nogle unikke karakterer eller strukturer, og kan derfor være 100% procedural. Der er stadig en måde at klare spillet, i form af at dræbe Ender Dragon, men strukturen der gør det muligt, bliver skabt rigtig mange gange, og der er derfor altid mulighed for at gøre det, lige meget hvor i verdenen, man er.

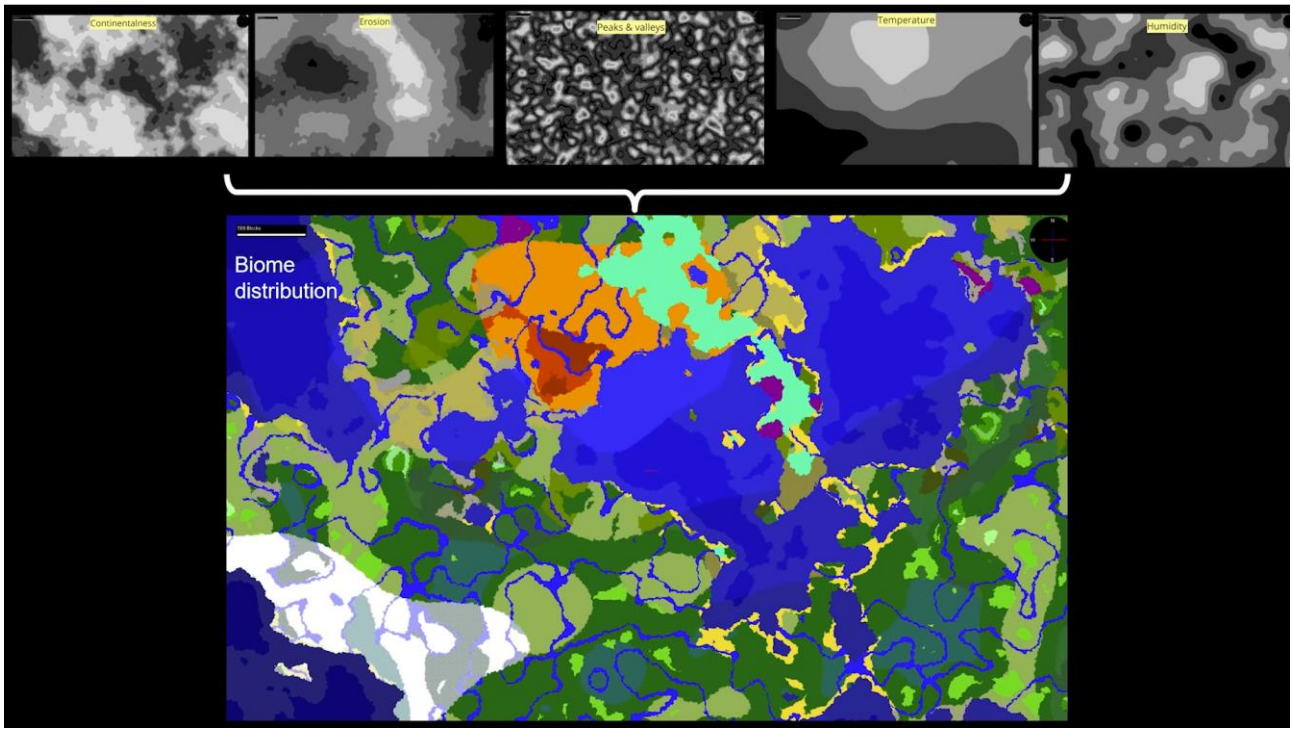
Udforskning

Hvis vi kigger på to spil som BotW og Minecraft, er udforskning en stor del af begge disse spil, men de opnår denne form for udforskning på to vidt forskellige måder, selvfølgelig fordi det ene er håndlavet og det andet er proceduralt.

I BotW er landmærkerne den drivende kraft for at få spilleren til at gøre noget. Hele deres verden er principielt bygget op på fundamentet af udforskning, og verdenen er derfor lavet specifikt til at spilleren konstant finder noget nyt og spændende. Denne tilgang er kun mulig, da verdenen er håndlavet, da man kan gå meget mere i detalje med hvor specifikke vigtige steder står, hvor meget spilleren kan se ved et givent punkt i verdenen og så meget mere. I en procedural verden er der meget mindre finjustering, da det er mere op til held, hvilke strukturer der står hvor. Hvis man kigger på trekants-reglen som BotW anvender, vil det være nær umuligt at gøre det samme i en procedural sammenhæng, da mængden af regler vil være uhåndgribeligt at gøre som en algoritme. For at det skulle fungere vil det kræve at alle bjerge var skabt med tanken om spillerens startposition, og at strukturer vil generere bag bjergene. På en mindre skala, kunne det fungere, men hvis det skulle vare næsten uendeligt, vil det være svært at lave noget unikt for hver formation af trekanter og strukturer.

[1] https://nomanssky.fandom.com/wiki/Artemis
[2] https://nomanssky.fandom.com/wiki/Space_Anomaly

For Minecraft har de taget et andet greb på denne opgave. Når man går rundt i verdenen, er der 5 forskellige værdier som bestemmer hvordan terrænet skal se ud. Det er continentalness, erosion, peaks & valleys, temperature og humidity. De har alle deres eget noisemap som bestemmer hvad værdien skal være ved et givent punkt i verdenen. De kan ses her



Figur 8 Illustration af de forskellige noises der bliver brugt til terræn generation, fra https://youtu.be/CSa5O6knuwI?si=_EjL0MUD3Fm-JJQh&t=1468, 24:28

Med disse værdier har generatoren en ide om hvordan terrænet ser ud, og derved hvordan de forskellige biomes[3] skal hænge sammen. For eksempel vil den sjældent sætte et meget varmt biome tæt på et rigtig koldt biome, som også er illustreret på billedet ovenfor. Her kan vi direkte se hvordan temperature noisemappet har gjort at de varmere biomes er midten-øverst af billedet, og de koldere dele er primært i venstre nederste hjørne.

Det samme kan ses med de andre værdier, for eksempel bestemmer continentalness direkte, hvor der er hav, og hvor der er land. Med denne tilgang er der et lag af realisme og sammenhæng i ens udforskning, men stadig en mangel på begrundelse bag placeringen af strukturer og biomes. Dette gør at under udforskningen af verdenen er der ikke nogen bevidst satte strukturer som drager din opmærksomhed på et brødkrummespor, ligesom der er i BotW,

[3] https://minecraft.fandom.com/wiki/Biome

tværtimod er det mere op til tilfældighed. Dette gør, at de ting du udforsker i spillet, ikke har den samme effektive teknik som i en håndlavet verden, men til gengæld giver det langt mere mulighed for replayability (muligheden for at spille spillet flere gange). Hvis verdenen er procedural, vil der altid være nyt og unikt terræn som aldrig er blevet set før, hvilket åbner op for rigtig mange muligheder hver gang du spiller spillet, lige meget hvor mange gange du har gjort det før.

Statiske og dynamiske verdener

Hvis vi kigger på en håndlavet verden som BotW, er elementer i den en del mere statisk end f.eks Minecraft. Der er monstre, som du kan dræbe, træer du kan fælde, sten som du kan ødelægge, men alle disse ændringer bliver nulstillet efter ikke så lang tid. Langt størstedelen af terrænet vil forblive det samme igennem hele spillet, og derved også hver gang du genspiller det. Dette er et svagere punkt for håndlavede verdener, da verdenen ofte ender med at være meget statiske, og hvis noget skal ændre sig imens man spiller, er det ofte scripted[4]. Hvis man kigger på håndlavede verdener i spil som Subnautica og Satisfactory, er de begge klassificeret som en form for survival spil, som ofte har dynamiske verdener. I deres tilfælde kan man ændre verdenen ved at bygge bygninger ovenpå den, men den fundamentale terrænstruktur kan ikke ændres. Dette gør at en spiller der genspiller spillet, allerede vil kende til verdenen. Derimod, i et spil som Minecraft, har dens procedurale terræn gjort at de kunne gøre hver eneste block ødelæggelig. Det gør at spilleren har langt mere kontrol over dynamikken at terrænet, hvilket leder til meget mere indviklet og dybdegående bygninger lavet af spilleren, som giver spilleren mulighed for at folde deres kreativitet ud. Denne ekstremt dynamiske verden er en af grundene til Minecrafts succes.

[4] En sekvens i et spil som er lavet af udviklerne til at ske på præcis den samme måde, ofte med meget lidt spiller interaktion, f.eks en cutscene

Figur 9 Eksempel på spiller-lavet bygninger i Minecraft, lavet af shovel241 https://www.youtube.com/@shovel241

Udviklings omkostninger

Procedural generation er ofte set i mindre spilfirmaer, da omkostningerne bag et 100% håndlavet terræn er rigtig store. Valget at gøre deres spil proceduralt åbner op for mange flere muligheder til at lave større spil, uden mange omkostninger bag det. Dette kan vi se i spil som No Man's Sky, hvor procedural generation har gjort at de kunne simulere et helt univers med 18 kvintillioner planeter (18,446,744,073,709,551,616). Dette har gjort at de kunne fokusere på andre aspekter af spillet. Såsom at give spilleren mulighed for at ødelægge og bygge på terræn der er skabt, samt at lave baser på de forskellige planeter og håndtere ressourcer der er proceduralt genereret fra dem. I tilfældet af Minecraft, har det gjort at de har kunne sætte mere vægt på spillerens påvirkning på verdenen.

Konklusion

De forskellige to måder at lave verdener til sine spil har deres styrker og ulemper alt efter hvilket slags spil man skal lave. Hvis ens mål primært ligger på at spilleren skaber sin egen narrativ ud fra de redskaber de er blevet givet, er det ofte en procedural verden der bliver anvendt.[5] Omvendt, hvis et spil er stærkt historiebaseret, er det ofte en del nemmere at skabe sit

[5] F.eks. Minecraft, No man's Sky, Terraria, Don't Starve,

eget håndlavet terræn hvor de forskellige aspekter kan blive brugt som storytellings-værktøjer i form af scripted-events eller lignende.[6]

Procedural Verdensgeneration demo

For at lave en dybdegående forklaring af procedural generation vil jeg skabe en demo som er baseret på Minecrafts verdensgeneration. Den er baseret på disse to tutorial serier https://www.youtube.com/playlist?list=PLcRSafycjWFdYej0h_9sMD6rEUCpa7hDH, https://www.youtube.com/playlist?list=PLcRSafycjWFesScBq3JgHMNd9Tidvk9hE. Jeg vil primært fokusere på den procedurale generation aspekt af demoen, og derfor vil nogle dele, som for eksempel, at oprette blocks/voxels, ikke blive beskrevet. Demoen er udviklet i Unity, og alle scripts er derved lavet i C#.

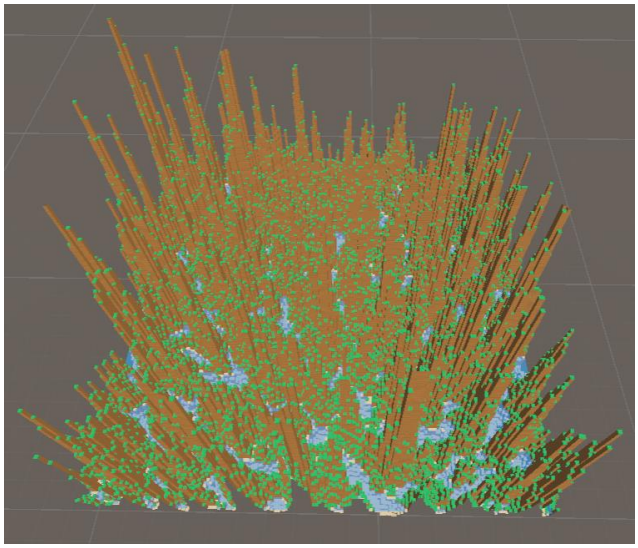
Generel brug og modifikation NoiseSettings

Programmet kan skabe et terræn ud fra Perlin noise, hvor man kan redigere i de forskellige værdier ved brug af ”Noise Settings” til at indstille zoom niveau, oktaver, persistance, redistribution modifier og exponent. De styrer forskellige dele af hvordan vores Perlin noise genereres, og hvordan det læses.

Noise zoom

Noise zoom justerer hvor langt ind, der skal zoomes på det Perlin noise som bliver genereret, siden det noise, der bliver lavet, er meget stor, skal man bruge meget små værdier for at det terræn der bliver generet, giver mening at bruge som verden.

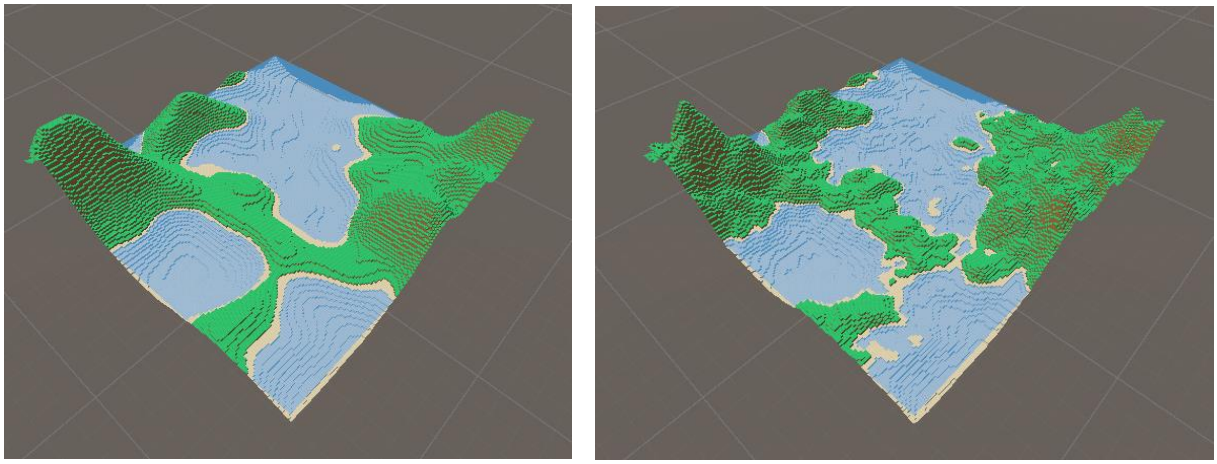
[6] F.eks. Red Dead Redemption 2, The Elder Scrolls V: Skyrim, God of War (2018), Sekiro: Shadows Die Twice



Figur 10 Terræn med zoom på 0,1

Oktaver

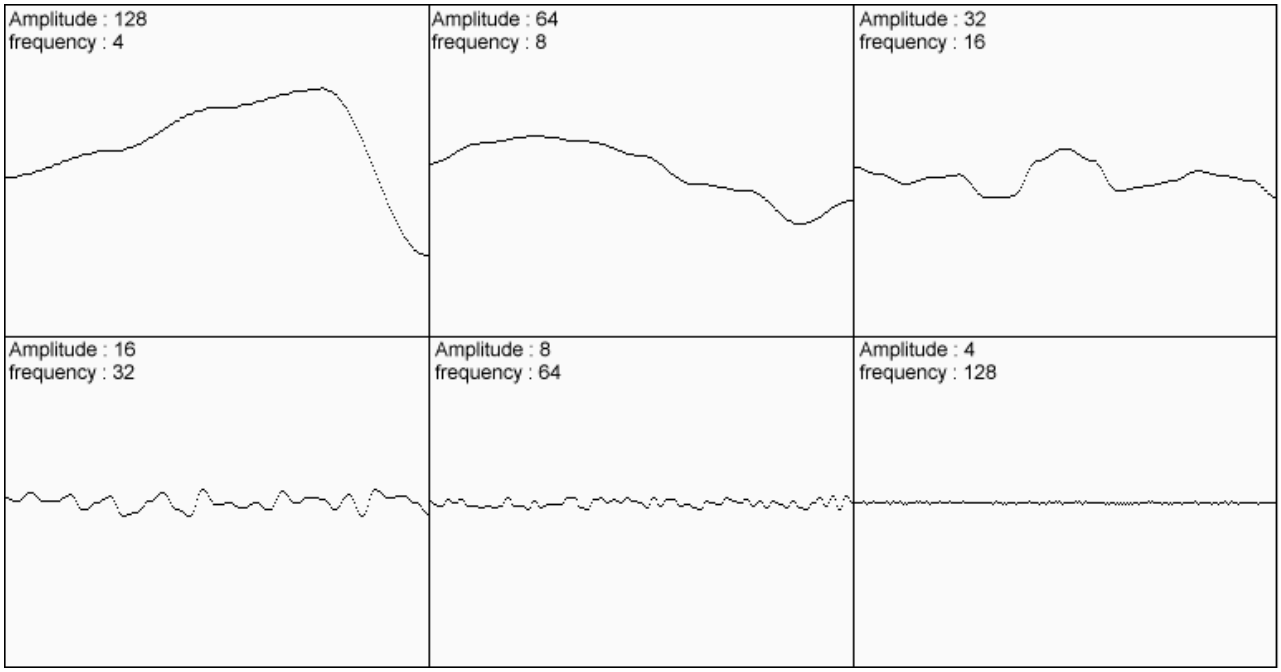
Oktaver beskriver kompleksiteten af vores Perlin noise, altså hvor mange elementer der er i det. Dette kan afbildes i det terræn der bliver genereret.

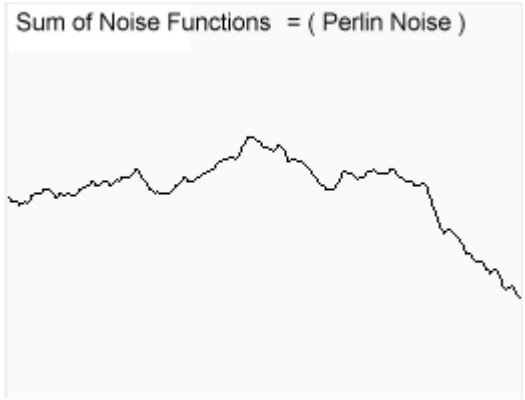


Figur 11, 3 Oktaver (venstre) og 5 oktaver (højre)

Vi ser at det terræn med færre oktaver har mindre kompleks, og mere afrundet terræn.

Oktaver kan beskrives som det antal lag af forskellige Perlin noises der bliver lagt sammen. Dette kan vises ved at kigge på 1D Perlin noise. Hvis vi har 6 forskellige Noises af varierende amplitude og frekvens, kan summen af dem give et mere realistisk terræn.

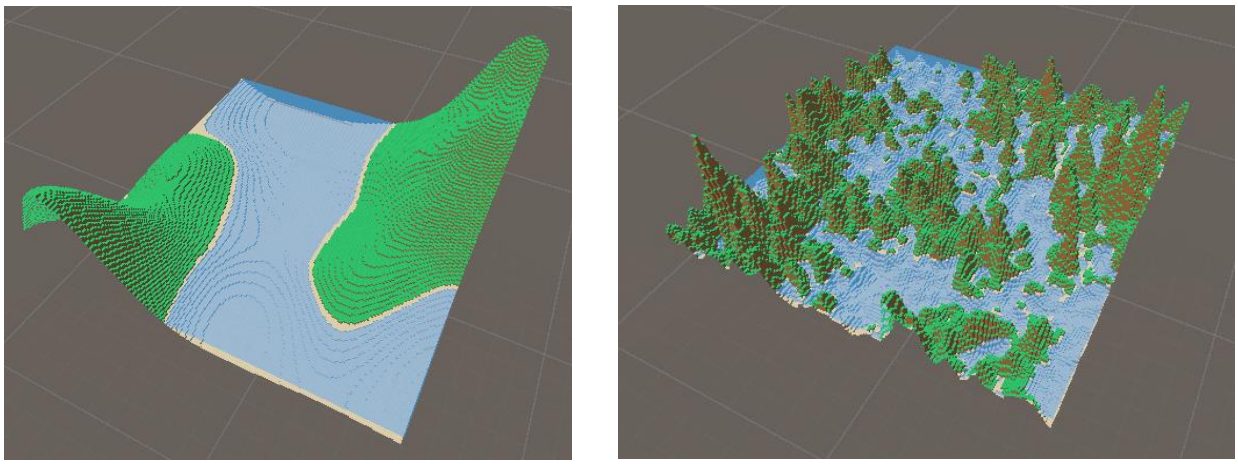




Figur 12 Tabel over 6 forskellige Perlin noises samt summen af dem, fra https://adrianb.io/2014/08/09/perlinnoise.html

Persistence

Persistence beskriver hvor meget indflydelse hvert lag(oktav) har. Ved højere tal resulterer det i terræn med hurtigere frekvenser, altså at det hurtigere går op og ned. Derimod giver en lavere værdi større, men mindre forekommende svingninger i terrænet.



Figur 13 Persistance på 0.1 (venstre) og 1 (højre)

Redistribution modifier og exponent

Beskriver amplituden, altså højden af terrænet, exponent er derved bare eksponenten til redistribution modifier, hvilket vi også kan se i MyNoise. Ændring af exponent resulterer derved også i en ændring af terrænets højde.

```
22.     public static float Redistribution(float noise, NoiseSettings settings)
23.     {
24.         return Mathf.Pow(noise * settings.redistributionModifier, settings.exponent);
25.     }
```

Kodeforklaring (Terrænmodel)

Når terrænet bliver genereret, følger den denne generelle struktur igennem de forskellige scripts.

World.cs(GenerateWorld) -> TerrainGenerator.cs(GenerateChunkData) -> BiomeGenerator.cs (ProcessChunkColumn) -> Chunk.cs (SetBlock)

At beskrive trinnene i denne rækkefølge giver derved en øget forståelse for strukturen af programmet.

World.cs

Chunks

Spillet bliver delt op i disse chunks for bedre at håndtere optimeringen af spillet. Hvis vi for eksempel tog Minecraft, som gør brug af samme chunk-system, har det gjort for dem, at spilleren gradvist afslører nye chunks af verdenen som de bevæger sig rundt. Det vil sige, at de chunks, som oprindeligt var synlige, da spilleren var i dens oprindelige position, nu er usynlige, når spilleren bevæger sig væk fra dem. Ved at gøre det i større bidder kræver det færre udregninger for hver chunk at finde ud af præcis hvordan den skal se ud, ud fra den data den har fra Perlin noise. Hvis disse chunks er for små, vil det kræve at computeren laver mange flere udregniner, og derved have en negativ effekt på køretidskompleksiteten. Hvis de derimod er for store, vil det resultere i at for mange blocks bliver vist for spilleren, som er unødvendige. En størrelse på 16 er derved en god balance mellem antal af udregninger, og antal af blocks der bliver vist

Forklaring

Dette script indeholder vores variabler såsom mapSizeInChunks, som beskriver hvor stort det genererede område skal være, samt chunkSize og chunkHeight som beskriver størrelsen af vores chunks.

Vi laver henvisninger til henholdsvis det prefab gameObject som skal fungere som vores chunk, og det script som skal være vores TerrainGenerator. Vi har også en vector som beskriver det seed der skal bruge i generationen af Perlin noise.[7]

```
8.      public int mapSizeInChunks = 10;
9.      public int chunkSize = 16, chunkHeight = 100;
10.
11.      public GameObject chunkPrefab;
12.
13.      public TerrainGenerator terrainGenerator;
14.      public Vector2Int mapSeedOffset;
```

Dette seed redigeres når man gerne vil have anderledes Perlin noise, hvis den ikke ændres, vil det samme terræn genereres hver gang.

[7] World.cs, linje 8-14

GenerateWorld begynder med at fjerne alle tidligere registreret chunks i vores chunkDataDictionary. De tidligere registrerede chunks skal ikke bruges længere, når verdenen regenereres.[8]

```
21.         chunkDataDictionary.Clear();
22.         foreach (ChunkRenderer chunk in chunkDictionary.Values)
23.         {
24.             Destroy(chunk.gameObject);
25.         }
26.         chunkDictionary.Clear();
```

Vi laver et for-loop med et nested for-loop inde i det. De laver iteration over henholdsvis x og z, indtil de værdier er større end vores mapSizeInChunks. I Unity er y-koordinatet højden, x og z er derfor længden og bredden af vores chunk. For hver nested iteration laver den et nyt chunk ud fra vores chunkSize og Height, samt koordinaterne som det chunk skal være ved, som er bestemt ud fra vores iterations variabler x og z. Den laver derefter et funktionskald til GenerateChunkData som håndterer selve generationen af vores chunk, derved hvordan det skal se ud i relation til vores Perlin noise. Den nye data om det genererede chunk bliver tilføjet til chunkDataDictionary med dens position og data som key og value, hvor key er en Vector3Int.[9]

```
28.         for (int x = 0; x < mapSizeInChunks; x++)
29.         {
30.             for (int z = 0; z < mapSizeInChunks; z++)
31.             {
32.
33.                 ChunkData data = new ChunkData(chunkSize, chunkHeight, this, new Vector3Int(x *
chunkSize, 0, z * chunkSize));
34.                 ChunkData newData = terrainGenerator.GenerateChunkData(data, mapSeedOffset);
35.                 chunkDataDictionary.Add(newData.worldPosition, newData);
36.             }
37.         }
```

Efter vores data er oprettet i chunkDataDictionary bruges et foreach-loop til at kigge i vores value til dette dictionary, og oprette MeshData, et nyt GameObject og en ChunkRenderer for hvert chunk. Dette gør at et gameobject bliver skabt inde i Unity for hvert chunk, og de unikke egenskaber for rendering bliver tilføjet til det.[10]

```
39.         foreach (ChunkData data in chunkDataDictionary.Values)
```

[8] World.cs, linje 21-26
[9] World.cs, linje 28-37
[10] Worlds.cs linje 39-47

```
40.         {
41.             MeshData meshData = Chunk.GetChunkMeshData(data);
42.             GameObject chunkObject = Instantiate(chunkPrefab, data.worldPosition, Quaterni-
on.identity);
43.             ChunkRenderer chunkRenderer = chunkObject.GetComponent<ChunkRenderer>();
44.             chunkDictionary.Add(data.worldPosition, chunkRenderer);
45.             chunkRenderer.InitializeChunk(data);
46.             chunkRenderer.UpdateChunk(meshData);
47.         }
```

Herinde skabes også en hjælpe-metode ”GetBlockFromChunkCoordinates” som anvendes inde i Chunk.cs. Den udregner først positionen af chunket, og laver et forsøg på at få data fra den position, som den gemmer i containerChunk. Hvis der ikke eksisterer noget ved det chunk forbliver containerChunk Null.[11]

```
50.     internal BlockType GetBlockFromChunkCoordinates(ChunkData chunkData, int x, int y, int z)
51.     {
52.         Vector3Int pos = Chunk.ChunkPositionFromBlockCoords(this, x, y, z);
53.         ChunkData containerChunk = null;
54.
55.         chunkDataDictionary.TryGetValue(pos, out containerChunk);
56.
57.         if (containerChunk == null)
58.             return BlockType.Nothing;
```

Den lokale position af den block inde i chunket bliver udregnet og gemt i blockInChunkCoordinates variablen. Til sidst returneres funktionskaldet Chunk.GetBlockFromChunkCoordinates med de to variable containerChunk og blockInChunkCoordinates.[12]

```
59.         Vector3Int blockInChunkCoordinates = Chunk.GetBlockInChunkCoordinates(containerChunk, new
Vector3Int(x, y, z));
60.         return Chunk.GetBlockFromChunkCoordinates(containerChunk, blockInChunkCoordinates);
```

TerrainGenerator.cs og BiomeGenerator.cs

TerrainGenerator

TerrainGenerator er et simpelt script der afhænger af BiomeGenerator til at generere data for hvert chunk, som scripts som World.cs bruger til at generere de chunks som gameObjects.

Den har kun en funktion, GenerateChunkData, der tager parametrene data og mapSeedOffset. Den bruger på samme måde som i World.cs et nested for-loop til at gentage over vores x og z

[11] World.cs, linje 50-58
[12] World.cs, linje 59-60

koordinater, og returnerer data for hver chunk. ”data” variablen bliver givet ved funktionskaldet biomeGenerator.ProcessChunkColumn, som tager parametrene data, x, z og mapSeedOffset.[13]

```
 6. public class TerrainGenerator : MonoBehaviour
 7. {
 8.     public BiomeGenerator biomeGenerator;
 9.     public ChunkData GenerateChunkData(ChunkData data, Vector2Int mapSeedOffset)
10.     {
11.         for (int x = 0; x < data.chunkSize; x++)
12.         {
13.             for (int z = 0; z < data.chunkSize; z++)
14.             {
15.                 data = biomeGenerator.ProcessChunkColumn(data, x, z, mapSeedOffset);
16.             }
17.         }
18.         return data;
19.     }
20. }
```

BiomeGenerator

Hvis vi kigger i BiomeGenerator ser vi hvordan ProcessChunkColumn fungerer. Herinde starter vi med at definere højden for der hvor vores vand skal genere, det vil sige at alle tomme block-pladser under dette koordinat, vil blive til vand. Derefter indlæser vi vores NoiseSettings, som bliver beskrevet i ”Generel brug og modifikation af NoiseSettings”. ProcessChunkColumn tager parametrene data, x, z og et mapSeedOffset. Det offset bliver med det samme sat til at være lig med værdien angivet i NoiseSettings. Variablen groundPosition bliver sat til returværdien af GetSurfaceHeightNoise med parametrene data.worldPosition.x + x, z + data.worldposition.z og data.chunkHeight. Ved at addere deres globale position i verdenen (data.worldPosition) med det lokale koordinat (x eller z) får vi det korrekte koordinat hvor vores noise skal blive brugt.[14]

```
6. public class BiomeGenerator : MonoBehaviour
7. {
8.     public int waterThreshold = 50;
9.     public NoiseSettings biomeNoiseSettings;
10.     public ChunkData ProcessChunkColumn(ChunkData data, int x, int z, Vector2Int mapSeedOffset)
11.     {
```

[13] TerrainGenerator.cs linje 6-20
[14] BiomeGenerator, linje 6-13

```
12.         biomeNoiseSettings.worldOffset = mapSeedOffset;
13.         int groundPosition = GetSurfaceHeightNoise(data.worldPosition.x + x, z + da-
ta.worldPosition.z, data.chunkHeight);
```

GetSurfaceHeightNoise

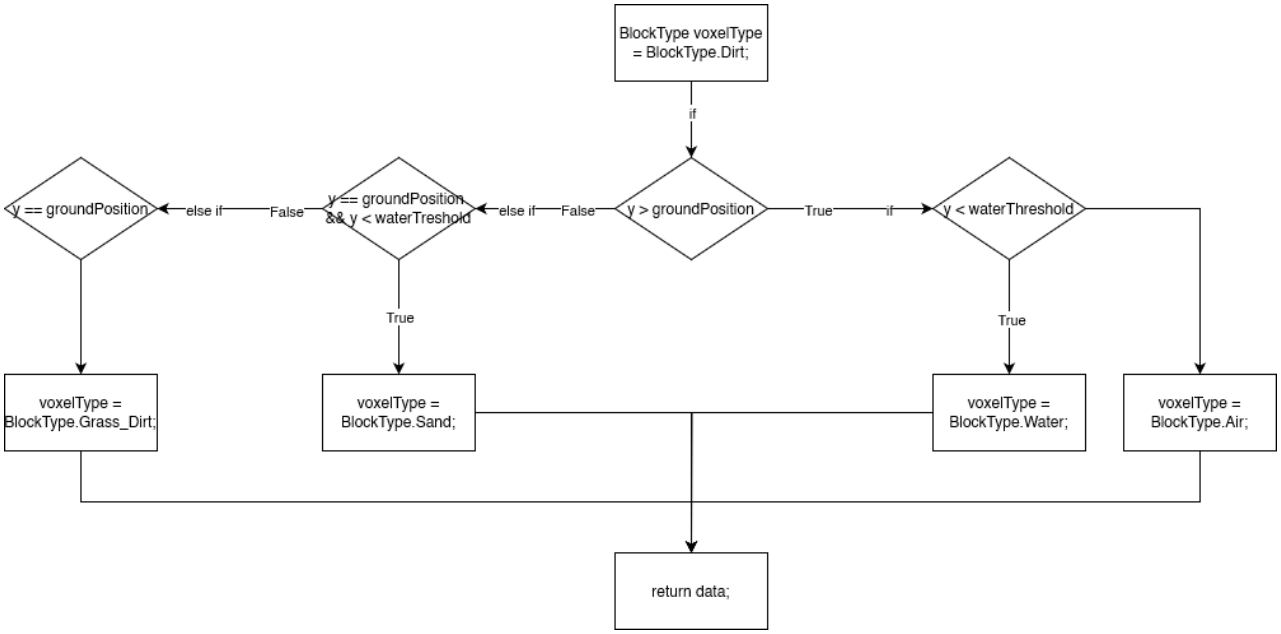
Hvis vi kigger i funktionen GetSurfaceHeightNoise kan vi se hvordan disse parametre bliver brugt. Først laver vi en variabel ud fra returværdien af MyNoise.OctavePerlin der bruger vores x og z koordinat samt biomeNoiseSettings. Variablen terrainHeight bliver sat til returværdien af MyNoise.Redistribution, og surfaceHeight til værdien af MyNoise.RemapValue01ToInt. Til sidst returneres surfaceHeight.[15]

```
43.     private int GetSurfaceHeightNoise(int x, int z, int chunkHeight)
44.     {
45.         float terrainHeight = MyNoise.OctavePerlin(x, z, biomeNoiseSettings);
46.         terrainHeight = MyNoise.Redistribution(terrainHeight, biomeNoiseSettings);
47.         int surfaceHeight = MyNoise.RemapValue01ToInt(terrainHeight, 0, chunkHeight);
48.         return surfaceHeight;
49.     }
```

De 3 funktioner til disse funktionskald forklares yderligere i MyNoise.cs.

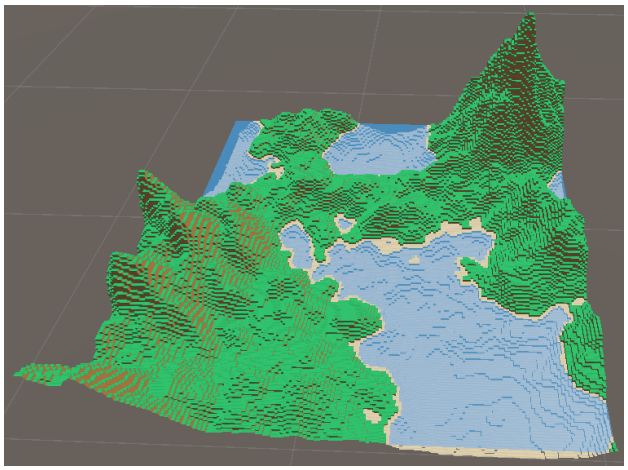
Hvis vi kigger tilbage ved linje 14 starter vi et for-loop der kører indtil y er større end vores data.chunkHeight.

[15] BiomeGenerator, linje 43-49



Figur 14 Flowchart over block logik

Den starter med at sætter vores voxelType til at være lig med BlockType.dirt, da det er vores standard. Efter det Anvender vi en masse if-statements til at vurdere hvilken type block/voxel der skal placeres. Hvis y er større end vores nuværende groundPosition går den et lag dybere og tjekker om den er mindre end vores waterThreshold, hvis ja bliver vores block sat til vand, og ellers bliver det til luft. Hvis y ikke er større end groundPosition tjekker den om den er præcis lig med groundPosition og om den er mindre end vores waterThreshold, hvis ja, bliver vores block til sand. Dette kan ses i vores genereret terræn, da alle blocks som er på niveau med, eller under vandet, bliver til sand.



Figur 15 Genereret terræn fra demo

Det sidste check, er for at se, om y er præcis lig med vores groundPosition, da der i så fald skal være en græs block, fremfor jord.

Chunk.cs

Chunk.cs indeholder en masse hjælpe-funktioner som bliver anvendt andre steder i programmet i relation til datahåndtering af chunks.

LoopThroughTheBlocks

Den første bruges til at gå igennem alle blocks i et chunk og udføre en aktion. Den skal derfor bruge chunkData som parameter samt en aktion som tager 3 integers som parameter. Her startes et for-loop der kører indtil alle blocks i vores chunkData er gået igennem. For hver block udfører den aktionen med de 3 positionelle koordinater.[16]

```
6.     public static void LoopThroughTheBlocks(ChunkData chunkData, Action<int, int, int> actionToPer-
form)
7.     {
8.         for (int index = 0; index < chunkData.blocks.Length; index++)
9.         {
10.             var position = GetPositionFromIndex(chunkData, index);
11.             actionToPerform(position.x, position.y, position.z);
12.         }
13.     }
```

Vi kan se et eksempel på en sådan aktion på linje 87. Her bliver LoopThroughTheBlocks kaldt med parametrene chunkData og x,y,z og et lambda-udtryk for at opdatere meshData med det nye chunk. Igen bliver to hjælpe-funktioner brugt her, GetMeshData og GetIndexFromPosition.

```
87.         LoopThroughTheBlocks(chunkData, (x, y, z) => meshData = BlockHelper.GetMeshData(chunkData,
x, y ,z, meshData, chunkData.blocks[GetIndexFromPosition(chunkData, x, y, z)]));
```

GetPositionFromIndex

En simpel funktion som der returnerer en Vector3Int ud fra et index. Den konverterer derfor et index tal til de korresponderende koordinater til den. Den gør dette ved at udregne x, y og z koordinaterne ud fra størrelsen og højden af det valgte chunk.[17]

```
15.     private static Vector3Int GetPositionFromIndex(ChunkData chunkData, int index)
16.     {
17.         int x = index % chunkData.chunkSize;
```

[16] Chunk.cs, linje 6-13
[17] Chunk.cs, linje 15-21

```
18.         int y = (index / chunkData.chunkSize) % chunkData.chunkHeight;
19.         int z = index / (chunkData.chunkSize * chunkData.chunkHeight);
20.         return new Vector3Int(x, y, z);
21.     }
```

For eksempel, er x koordinatet udregnet ved at tage vores index og tage den resterende værdi af dens division med chunkSize ved brug af modulo operatoren.

InRange

En kort funktion som vurderer hvorledes et koordinat er indenfor et chunks lokale koordinatsystem. Den tjekker først om tallet enten er under 0, eller om det er lig med, eller større end størrelsen af vores chunkSize, hvorledes den returnerer false hvis en af dem er rigtige.

```
24.     private static bool InRange(ChunkData chunkData, int axisCoordinate)
25.     {
26.         if (axisCoordinate < 0 || axisCoordinate >= chunkData.chunkSize)
27.             return false;
28.
29.         return true;
30.     }
```

InRangeHeight

Præcis den samme logik bliver anvendt her, bare med et y-koordinat og chunkHeight i stedet.

```
32.     private static bool InRangeHeight(ChunkData chunkData, int yCoordinate)
33.     {
34.         if (yCoordinate < 0 || yCoordinate >= chunkData.chunkHeight)
35.             return false;
36.
37.         return true;
38.     }
```

GetBlockFromChunkCoordinates

Denne funktion er overloaded, hvilket vil sige at den har to varianter med forskellige parametre, den første, fra linje 39-42, fungerer som en wrapper for den anden. Det vil sige at den første funktion anvender Vector3Int i dens parametre, mens den anden anvender 3 integers. Reelt gør det bare, at du kan kalde funktionen med både en Vector3Int eller 3 integers, alt efter hvilken datatype du arbejder med. Den anden funktion er derfor den som laver udregningerne

Først starter den med at tjekke om alle koordinaterne er indenfor vores range i chunkets lokale koordinatsystem. Hvis ja, sætter den variablen index til returværdien af GetIndexFromPosi-

tion med parametrene chunkData, x,y og z. Til sidst returnerer den blockdata indenfor det givne chunk ud fra den udregnede index.

Hvis ikke koordinaterne er indenfor vores range, bliver funktionen rekursivt kaldt igen under en worldReference, og ved at tilføje chunkData.worldPosition til alle koordinaterne, leder den efter den blocks position i hele verdens koordinater.[18]

```
45.     public static BlockType GetBlockFromChunkCoordinates(ChunkData chunkData, int x, int y, int z)
46.     {
47.         if (InRange(chunkData, x) && InRangeHeight(chunkData, y) && InRange(chunkData, z))
48.         {
49.             int index = GetIndexFromPosition(chunkData, x, y, z);
50.             return chunkData.blocks[index];
51.         }
52.         return chunkData.worldReference.GetBlockFromChunkCoordinates(chunkData,
chunkData.worldPosition.x + x, chunkData.worldPosition.y + y, chunkData.worldPosition.z + z);
53.     }
```

SetBlock

Denne funktion bruges til at sætte en block ved et givent punkt i de lokale koordinater af et chunk. Den starter med at tjekke validiteten af koordinaterne med samme metode som i de tidligere funktioner ved at bruge InRange og InRangeHeight til at tjekke om de givne koordinater er inden for vores chunk. Hvis de givne koordinater ikke opfylder det krav, smides der en exception. I en videreudviklet implementation til en større skala vil der være kode til at finde vores block indenfor verdenskoordinaterne, fremfor chunkets lokale koordinater. Men på en denne mindre skala, er det ikke nødvendigt.
Hvis vores koordinater er indenfor den lokale range, bliver index sat til returværdien af funktionen GetIndexFromPosition, og den bliver registreret i chunkData.blocks listen.[19]

```
55.     public static void SetBlock(ChunkData chunkData, Vector3Int localPosition, BlockType block)
56.     {
57.         if (InRange(chunkData, localPosition.x) && InRangeHeight(chunkData, localPosition.y) &&
InRange(chunkData, localPosition.z))
58.         {
59.             int index = GetIndexFromPosition(chunkData, localPosition.x, localPosition.y, localPo-
sition.z);
60.             chunkData.blocks[index] = block;
61.         }
```

[18] Chunk.cs, linje 45-53
[19] Chunk.cs, linje 55-66

```
62.         else
63.         {
64.             throw new Exception("Need to ask World for appropriate chunk");
65.         }
66.     }
```

GetIndexFromPosition

Denne funktion gør det omvendte af GetPositionFromIndex, idet at man indsætter sin chunk-Data samt x,y og z koordinater, og i retur, får man den resulterende index[20]

```
68.     private static int GetIndexFromPosition(ChunkData chunkData, int x, int y, int z)
69.     {
70.         return x + chunkData.chunkSize * y + chunkData.chunkSize * chunkData.chunkHeight * z;
71.     }
```

Man kan se processen af udregningerne hvis vi tager nogle tilfældige koordinater og laver beregningerne både til index og tilbage igen med dem. I dette eksempel er x = 10, y = 3 og z = 7. chunkSize og Height forbliver deres normale værdier på 16 og 100.

Koordinater til index:

Input

$10 + 16 \times 3 + 16 \times 100 \times 7$

Result

11 258

Index til x:

Input

$11\,258 \bmod 16$

Result

10

Index til y:

Input

$\frac{11\,258}{16} \bmod 100$

Exact result

$\frac{29}{8}$

Decimal form

3.625

Index til z

Input

$\frac{11\,258}{16 \times 100}$

Exact result

$\frac{5629}{800}$

Decimal form

7.03625

Da man altid runder ned ved beregning af integers, får vi derved de korrekte resultater 10, 3 og 7.

GetBlockInChunkCoordinates

Denne mindre funktion konverterer globale block koordinater til lokale chunk koordinater ved at subtrahere det givne blocks position med de globale koordinater af chunket.

```
73.     public static Vector3Int GetBlockInChunkCoordinates(ChunkData chunkData, Vector3Int pos)
74.     {
75.         return new Vector3Int
76.         {
```

[20] Chunk.cs, linje 68-71

```
77.             x = pos.x - chunkData.worldPosition.x,
78.             y = pos.y - chunkData.worldPosition.y,
79.             z = pos.z - chunkData.worldPosition.z
80.         };
81.     }
```

GetChunkMeshData

Denne funktion returnerer meshData for et givent chunk, som den bruger hjælpefunktionerne LoopThroughTheBlocks, GetMeshData og GetIndexFromPosition. Den laver først en ny variabel der hedder meshData og bruger, som tidligere nævnt i LoopThroughTheBlocks, et lambda-udtryk til at kalde LoopThroughTheBlocks. Dette giver den modificeret meshData tilbage, som nu indeholder det valgte chunk.

```
83.     public static MeshData GetChunkMeshData(ChunkData chunkData)
84.     {
85.         MeshData meshData = new MeshData(true);
86.
87.         LoopThroughTheBlocks(chunkData, (x, y, z) => meshData = BlockHelper.GetMeshData(chunkData,
x, y ,z, meshData, chunkData.blocks[GetIndexFromPosition(chunkData, x, y, z)]));
88.
89.         return meshData;
90.     }
```

ChunkPositionFromBlockCoords

Til sidst, har vi en funktion som der udregner positionen af et chunk ud fra en af dens tilhørende blocks. Den danner først en ny Vector3Int ved navn pos som bliver tilsat værdierne af x y og z. Disse værdier bliver beregnet ved at tage den afrundet værdi af x divideret med vores chunkSize og gange med chunkSize igen. Umiddelbart virker det udødvendigt, men ved at gøre dette sørger man for at der ikke sker nogle floating point errors i relation til de koordinater som bliver givet da de bliver afrundet til det nærmeste integer.

```
92.     internal static Vector3Int ChunkPositionFromBlockCoords(World world, int x, int y, int z)
93.     {
94.         Vector3Int pos = new Vector3Int
95.         {
96.             x = Mathf.FloorToInt(x / (float)world.chunkSize) * world.chunkSize,
97.             y = Mathf.FloorToInt(y / (float)world.chunkHeight) * world.chunkHeight,
98.             z = Mathf.FloorToInt(z / (float)world.chunkSize) * world.chunkSize
99.         };
100.         return pos;
111.     }
```

MyNoise.cs

RemapValue01

RemapValue01 bruges til at remap en given værdi mellem rangen 0 til 1 ud fra den valgte outputMin og outputMax. Den anvendes i BiomeGenerator.cs hvor den givne value er læst fra vores Perlin noise og er et tal mellem 0 og 1. Hvis vi anvender denne matematik på et af tallene får vi følgende

$$0 + (0.4277868 - 0) \times \frac{100 - 0}{1 - 0}$$

Result

42.77868

Figur 16 Resultat af RemapValue01 med tilfældig værdi

Essentielt bliver tallet bare ganget med 100, da vi ved hvert funktionskald har en outputMin på 0, og en outputMax på 100.

```
 8.     public static float RemapValue01(float value, float outputMin, float outputMax)
 9.     {
10.         return outputMin + (value - 0) * (outputMax - outputMin) / (1 - 0);
11.     }
```

RemapValue01Int

Denne funktion eksisterer kun for at være en hjælpe-funktion, den kalder blot RemapValue01 som en integer.

Redistribution

Denne funktioner tager to parametre, en float ”noise” og vores NoiseSettings og bruger den givne exponent og redistributionModifier til at returnere vores noise ganget med den valgte redistributionModifier og derefter opløftet i vores eksponent. På denne måde kan man nemt ændre i terrænets egenskaber ved at modificere disse to tal inde i NoiseSettings.

```
19.     public static float Redistribution(float noise, NoiseSettings settings)
20.     {
21.         return Mathf.Pow(noise * settings.redistributionModifier, settings.exponent);
22.     }
```

OctavePerlin

I OctavePerlin opretter vi den Perlin noise som vi anvender til vores terræn. Den bruger to parametre, x og z, der bestemmer de værdier der skal give til den Perlin noise der bliver genereret. Først bliver de givne værdier ganget med vores zoom niveau, for at sørge for at det noise der bliver genereret, er småt nok til at det kan anvendes som terræn. Vi danner nogle forskellige float variabler, de mest bemærkelsesværdige er amplitude og frequency, som beskriver amplituden og frekvensen af den Perlin noise vi genererer. "total" bliver brugt til at holde på den generede Perlin noise, så vi ved hver iteration af for-loopet kan tilføje endnu et lag af noise.

Vi laver et for loop hvor det kører indtil det overskriver antal settings.octaves. For hvert loop, bliver der genereret Perlin noise med PerlinNoise funktionen. Den første parameter er resultatet af det valgte offset på x-aksen lagt sammen med worldOffset.x og plusser det med x og til sidst ganger det med frequency. Ved at gøre dette får vi tilføjet det offset der er valgt, hvilket gør at man kan bevæge sig på akserne hvis man ændrer i denne offset værdi i NoiseSettings. Det samme gøres for den anden parameter, her ganges amplituden også sammen. Efter den første noise er genereret bliver amplitude adderet til amplitudeSum og ganget med vores persistence-værdi. Da den værdi ligger på 0.5 betyder det at alle fremtidige noises som bliver tilføjet har halv så stor amplitude som den tidligere. Samtidigt bliver frequency adderet med 2, og derved bliver frekvensen større for hver noise der bliver genereret.

Inden vores noise bliver returneret bliver det divideret med amplitudeSum for at normalisere de ændringer der er blevet lavet af amplituden, da det gør alle tallene meget høje. På denne måde bliver amplituderne af de forskellige noise-lag forskellige, uden at have for drastiske ændringer i terrænet.[21]

```
24.     public static float OctavePerlin(float x, float z, NoiseSettings settings)
25.     {
26.         x *= settings.noiseZoom;
27.         z *= settings.noiseZoom;
28.
29.         float total = 0;
30.         float frequency = 1;
31.         float amplitude = 1;
32.         float amplitudeSum = 0;  // Used for normalizing result to 0.0 - 1.0 range
```

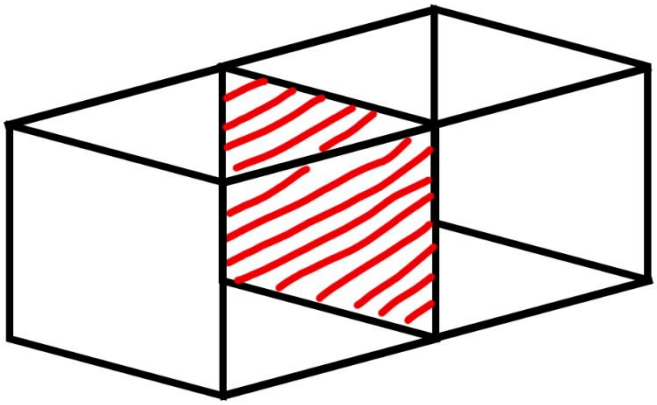
[21] MyNoise.cs linje 24-43

```
33.         for (int i = 0; i < settings.octaves; i++)
34.         {
35.             total += Mathf.PerlinNoise((settings.offset.x + settings.worldOffset.x + x) * frequen-
cy, (settings.offset.y + settings.worldOffset.y + z) * frequency) * amplitude;
36.
37.             amplitudeSum += amplitude;
38.
39.             amplitude *= settings.persistence;
40.             frequency *= 2;
41.         }
42.
43.         return total / amplitudeSum;
44.     }
```

Backface culling og optimering

Til dette program anvendes quads til at visualisere vores terræn. Det unikke ved denne opstilling, er at quads kun er synlige fra den ene side, vi kan derfor anvende dem til at lave backface culling på vores blocks. Det vil sige, at alle de blocks, som ikke er synlige, ikke bliver rendered af Unity. Vi kan se dette hvis man kigger under terrænet, og ser at det kun er de yderste og synlige sider der bliver vist.

Vi kan se dette i denne visualisering af 2 blocks ved siden af hinanden. Den midterste side, som ikke vil være synlig for spilleren, bliver derfor ikke rendered, og kræver derfor færre ressourcer fra computeren.



Figur 17 Visualisering af backface culling

Vurdering af terrænets egenskaber

Terrænet, der er genereret, har en semi-realistisk variation i dens højde, og der bliver derfor dannet sand, søer og bjerge af varierende højde. Det vil sige, at du som spiller vil opleve en relativ stor variation i det terræn der er blevet genereret. Tilgengæld er der ikke implementeret forskellige biomes, træer, græs, strukturer osv. Hvilket er meget afgørende egenskaber af procedural terræn. Det vil sige, at efter ikke særlig lang tid, vil terrænet blive ret kedeligt at udforske for en spiller. Som et udgangspunkt, fungerer terrænet som en procedural generator, men for at det skulle bruges i en spilindstilling, skulle der tilføjes langt mere detalje, for at det vil være det værd, at udforske for spilleren. Viderearbejdelse med denne demo vil derfor indeholde biomegeneration samt mere detalje i form af flora og fauna osv.

Konklusion

Ud fra undersøgelsen af procedural generation i relation til spil har det vist sine brugssager som værktøj til spiludvikling. Procedural generation bliver brugt i spil primært når en spiludvikler skal lave et realistisk terræn som der er uendeligt og unikt, så spilleren altid kan udforske noget nyt. Dette vises igennem spil som Minecraft og No Man's Sky. I modsætning kan et håndlavet terræn byde en mere udvalgt måde at lave historiefortælling, hvor alle hændelser er skabt at udvikleren for at fortælle en historie. Som spiludvikler skal man tage valget når man laver sin verden, da valget er ekstremt betydningsfuldt for hvordan spillets gameplay kommer til at hænde. Procedural terræn medfører ofte også en øget spillerinteraktion med verdenen, da det ofte kræver spillerens egen indsats at gøre verdenen tilpasset til dem. Dette kan ses i form af f.eks. at bygge huse i Minecraft. Derimod bliver verdenen sjældent ændret i håndlavede verdener, medmindre det er for at fortælle en historie. Procedurale og håndlavede verdener har derfor deres egne egenskaber som er nødvendige at forstå, for at kunne tage den rette beslutning til sit spil.

Litteraturliste

Kniberg, H. (9. maj 2022). *Youtube*. Hentet 11-19. december 2023 fra Reinventing Minecraft world generation by Henrik Kniberg: https://www.youtube.com/watch?v=ob3VwY4JyzE

Kniberg, H. (6. februar 2022). *Youtube*. Hentet 11-19. december 2023 fra Minecraft terrain generation in a nutshell: https://www.youtube.com/watch?v=CSa5O6knuwI

Toolkit, G. M. (9. maj 2023). *Youtube*. Hentet 11-19. december 2023 fra How Nintendo Solved Zelda's Open World Problem: https://www.youtube.com/watch?v=CZzcVs8tNfE

Walker, M. (2. october 2017). *Twitter*. Hentet 11-19. december 2023 fra Twitter: https://web.archive.org/web/20171003165205/https://twitter.com/gypsyOtoko/status/915037712507219969

Bilag

Større filer findes under ekstramateriale.

Perlin noise.py

```
 1. from PIL import Image
 2. import numpy as np
 3. from perlin_noise import PerlinNoise
 4. import matplotlib.pyplot as plt
 5.
 6. noise = PerlinNoise(octaves=10, seed=1)
 7.
 8. xpix, ypix = 100, 100
 9. pic = [[noise([i/xpix, j/ypix]) for j in range(xpix)] for i in range(ypix)]
10.
11. image_path = 'noise.png'
12. plt.imsave(image_path, pic, cmap="gray")
13.
14. #Convert to grayscale for number simplifaction
15. img = Image.open(image_path).convert('L')
16. perlin_array = np.array(img)
17.
```

```
18. list = []
19. for x in perlin_array:
20.     list.append(x[0])
21.
22. f = open("perlinarray.txt", "w+")
23. f.write(str(list))
24. f.close()
```