

PRESTANDAJÄMFÖRELSE AV BRUSALGORITMER

Hur presterar olika brusimplementationer i förhållande till en realtidsbaserad lösning?

PERFORMANCE COMPARISON OF NOISE ALGORITHMS

How does different noise algorithms perform in relation to a realtime-based solution?

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2026

Hannes Purmonen
Patrik Kroon

Handledare: Peter Sjöberg
Examinator: Mikael Thieme

Sammanfattning

Inom spelutveckling brukas procedurellt brus frekvent för terränggenerering delvis för att minimera mängden manuellt arbete. Vidare är algoritmerna deterministiska och har korta beräkningstider vilket är passande för realtidsbaserad generering. Detta arbete ämnar studera och jämföra prestandan för brusalgoritmerna *Perlin Noise*, *Simplex Noise* och *Gabor Noise* utifrån olika implementationer av dessa algoritmer. Experimentet sker sekventiellt på CPU:n utifrån flera upplösningar på två olika hårdvaror. En begränsad mängd verk studerar brusalgoritmernas prestanda men sällan inom sammanhanget av CPU:n. Från resultatet av studien blir det komplicerat att definiera generaliseringar till algoritmerna samt att implementationerna har en betydande påverkan. Övergripande presterar *Simplex Noise* med lägst exekveringstid medan *Gabor Noise* har en markant högre exekveringstid. Framtida arbete kan vidareutveckla denna studie genom att utforska flera aspekter såsom att undersöka parallellisering och undersöka flera dimensioner.

Nyckelord: Brus, CPU, Prestanda, Procedurell generering, Unity.

Innehållsförteckning

1	Introduktion	1
2	Bakgrund	2
2.1	Realtidsgenerering	2
2.2	Procedurellt Genererat innehåll	2
2.3	Procedurell terränggenerering	2
2.4	Brusalgoritmer	3
2.5	Lattice Gradient Noise & Sparse Convolution Noise	5
2.6	Perlin	5
2.6.1	Algoritm	5
2.6.2	Visuell karaktär	6
2.7	Simplex	7
2.7.1	Algoritm	7
2.7.2	Visuell karaktär	8
2.8	Gabor	9
2.8.1	Algoritm	9
2.8.2	Visuell karaktär	10
2.9	Tidigare arbeten	10
3	Problemformulering	12
3.1	Hypotes	12
3.2	Metodbeskrivning	12
3.3	Metoddiskussion & problematisering	13
3.3.1	Anpassningar och val	13
3.3.2	Upplösningar	14
3.3.3	Valet av CPU-implementation	14
3.3.4	Val av algoritmer och avgränsningar	15
4	Genomförande	17
4.1	Hanteringen av brusimplementationer	17
4.2	Artefakt	18
4.2.1	Stopwatch	19
4.3	Insamling och hantering av data	20
4.4	Resultat	20
4.4.1	Exekveringstid per upplösning	20
4.4.2	Median	26

4.5 Analys.....	28
5 Sammanfattning och diskussion	30
5.1 Sammanfattning.....	30
5.2 Diskussion	30
5.3 Samhälleliga och etiska aspekter.....	33
5.4 Framtida arbete	33
Referenser.....	35

1 Introduktion

Brusalgoritmer är matematiska funktioner som producerar till synes slumpmässigt men varierat brus (eng. noise). Utbudet av brusalgoritmer är stort och de olika varianterna utvecklas oftast för specifika användningsområden. Brusalgoritmer är vanligt förekommande inom diverse steg vid skapandet av spel. Ett vanligt implementeringssätt för brus är att generera bruskartor som sedan kan användas för att exempelvis skapa slumpmässig terräng. Detta är vad som gör brus relevant för just realtidsbaserade implementationer. Inom procedurrell generering kan brusalgoritmer bidra med dynamiska och levande spelvärldar, samt med relativt låga exekveringstider jämfört med fysikbaserade implementationer. På grund av den generellt låga exekveringstiden används brusalgoritmer frekvent inom realtidsbaserade implementationer för GPU (eng. *graphics processing unit*) och CPU (eng. *central processing unit*).

Inom spel finns ett behov av att bibehålla en relativt låg exekveringstid. Beroende på hur snabba de olika implementationerna är kan exekveringstiden påverka lämpligheten för realtidsanvändning. Vidare är brusalgoritmer utvecklade utifrån olika ändamål och sammanhang, detta innebär variationer i tillvägagångssättet för att beräkna och konstruera respektive brus. Därtill kan det finnas ett värde i vetenskapen om vilka sammanhang som algoritmerna är lämpliga för.

Detta arbete genomförde ett experiment med olika upplösningar som problemstorlekar för att samla in exekveringstiden från olika brusimplementationer och exekverades på CPU:n. Av det stora utbudet av brusalgoritmer testades *Perlin Noise*, *Simplex Noise* och *Gabor Noise*. Målet med resultatet är att kunna jämföra hur vardera implementations exekveringstid förhåller sig till övriga implementationer, men också försöka tolka algoritmernas lämplighet för applicering i realtidsbaserade sammanhang. För att stärka kvaliteten i resultaten från experimentet genomfördes vissa anpassningar, bland annat användes flera iterationer per problemstorlek för samma implementation.

Varje implementation kontrolleras före experimentet via en egenkonstruerad komponent som visualiserar bruset i höjdkartor (eng. *heightmaps*). Det konstrueras också en testmiljö för att samla in exekveringstiden från implementationerna som sedan presenteras och analyseras för att besvara frågeställningen.

2 Bakgrund

Detta kapitel beskriver exempel på olika användningsområden för brus inom spel och i utvecklingen av spel, vidare introduceras specifika brusalgoritmer som har använts inom denna studie. I slutet av kapitlet presenteras en genomgång av tidigare arbeten som bidrar till en grund för detta arbete, samt kartlägger tidigare studier och resultat inom området.

2.1 Realtidsgenerering

Inom vissa användningsområden kan det vara nödvändigt att kunna generera innehåll i realtid. Realtid innebär att innehållet genereras för direkt bruk med låg fördröjning och med högt bildantal, också kallat *frames per second* (FPS). Vidare förväntas en låg påverkan av den upplevda prestandan som är mer bunden till fluktuationer i tiden mellan varje bild (Liu, Kuwahara, Scovell & Claypool 2023).

Realtidsgenerering används inte endast inom spelutveckling men förekommer främst inom procedurrell terränggenerering (eng. *procedural terrain generation*, PTG) då det ofta används för att skapa ny terräng när världen utforskas i ett spel. Ett av de största och mest ikoniska spelen som använder sig av detta är Minecraft (Mojang 2009).

2.2 Procedurellt Genererat innehåll

Procedurellt genererat innehåll (eng. *procedural content generation*, PCG) framkommer i en rad olika tillvägagångssätt och används för att kunna procedurellt generera en stor variation av innehåll utan att manuellt behöva skapa allt. PCG berör flertalet områden, allt ifrån animation (Li & Zhao 2012) och texturer (Maung, Shi, & Crawfis 2012) till terränggenerering (Büyüksar, Yıldız & Demirci 2024).

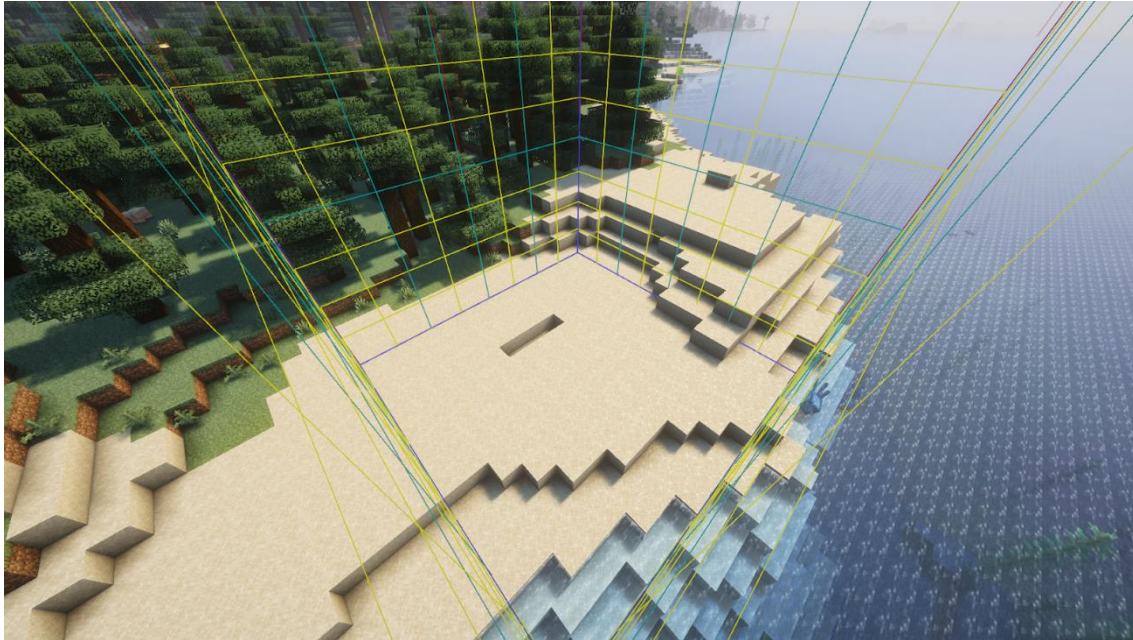
2.3 Procedurell terränggenerering

Det är frekvent återkommande att procedurell terränggenerering presenteras med brusalgoritmer. Argumentationen för att använda brus i detta område ligger i att det lätt kan användas för att skapa stora öppna världar med betydligt mindre manuellt arbete, som annars hade blivit mer tidskrävande (Fischer, Dittmann, Weller & Zachmann 2020; Zhang, Xu, Liu & Liu 2025).

Det framkommer i litteraturen att ett vanligt sätt att använda brusalgoritmer är för att konstruera en grund för terräng utifrån brus som sedermera kan utvidgas och förfinas. Denna ytterligare bearbetning kan sedan genomföras i kombination med andra processer som också kan vara grundade i brus (Büyüksar et al. 2024; Fischer et al. 2020).

Minecraft (2009) är ett tydligt exempel på en tillämpning av både realtidsgenerering och PTG. I en presentation av Henrik Kniberg (JFokus 2022) som tidigare har varit anställd på Mojang, beskriver han att spelet Minecraft genererar innehåll genom att dela upp spelvärlden i flertalet delområden (eng. *chunks*). Spelvärlden struktureras enligt ett rutnät bestående av delområden med statisk upplösning, specifikt $16 \times 16 \times 384$ i Minecraft. Varje enskilt område kan sedan beräknas separat från intilliggande grannområden. En andel av dessa beräkningar genomförs med olika brusalgoritmer i

flera delar av genereringsprocessen. Bruset kan appliceras på en tvådimensionell höjdkarta med en upplösning baserad på delområdets bredd och djup, för att främst avgöra höjdskillnader av en yta. Bruset begränsas inte till att endast generera höjdskillnader utan bidrar också till konstruktionen av geografiska miljöer för bland annat kullar, bergskedjor, hav och olika grottstrukturer i spelet.



Figur 1: Visualisering av chunks i spelet Minecraft (2009)

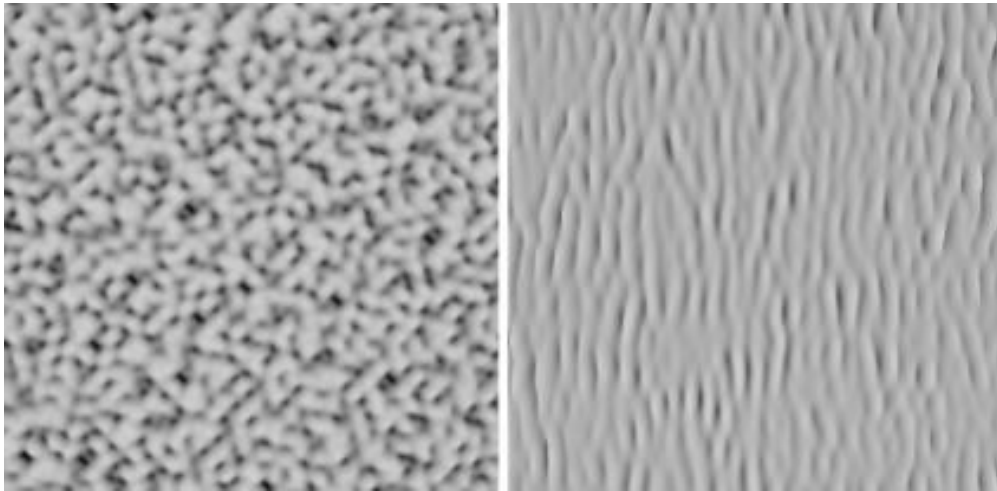
2.4 Brusalgoritmer

Brusalgoritmer är matematiskt grundade funktioner som producerar till synes slumpmässiga och varierande värden. Lagae et al. (2010) beskriver flera olika fördelar som generellt anses överensstämna med fullvärdiga brusalgoritmer. Dessa fördelar med generering av brus inkluderar bland annat att de bruskartor som genereras blir sammanhängande och även kan teoretiskt omfatta godtyckligt stora ytor. Algoritmerna har även generellt låg minnesanvändning vid beräkning av brusvärden och kan skapa värden i godtyckliga upplösningar och dimensioner. Ytterligare fördelar är möjligheten att kunna variera bruset som genereras via diverse parametrar. Avslutningsvis är det fördelaktigt att brusalgoritmer beräknar pixlar av brus med konstant tid. Den konstanta tiden gäller dock endast inom den dimension som algoritmen verkar i.

Vidare beskriver Lagae et al. (2010) att utbudet av olika brusalgoritmer är stort och att dessa olika algoritmer utvecklas oftast för specifika användningsområden. Brusalgoritmer kan därmed användas brett i olika steg för spelutveckling. Inom procedurrell generering kan brusalgoritmer bidra med dynamiska och levande spelvärldar, samt med relativt låga exekveringstider jämfört med fysikbaserade implementationer (Wang, Su, Qin & Wang 2025). En del av brusalgoritmer har även egenskapen att efterlikna fysikbaserade resultat, fast med oftast sämre visuella resultat (Galín et al. 2019). Fysikbaserade lösningar är baserade på fysiksimuleringar och resulterar vanligen i mer realistiska slutprodukter. Samtliga delar av detta har betydande relevans inom spelutveckling och spelindustrin (Galín et al. 2019) vilket syns på användandet av brusalgoritmer inom flertalet kända spel som till exempel Minecraft

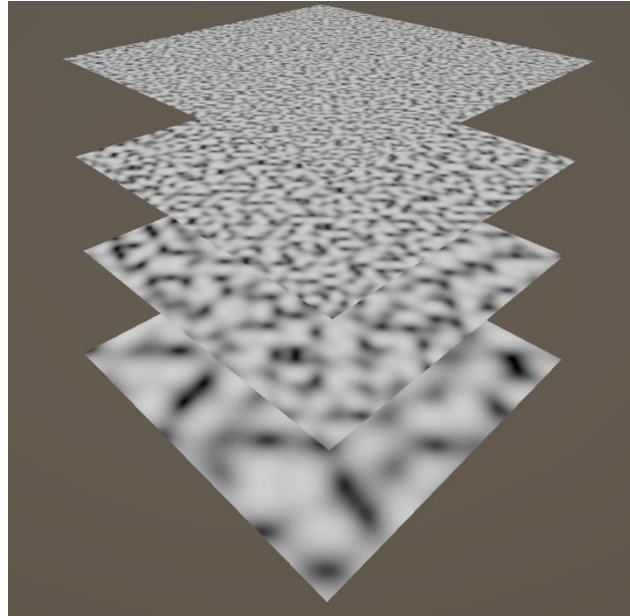
(Mojang 2009) och Terraria (Re-Logic 2011).

Brusalgoritmer kan grupperas utifrån egenskapen att generera isotropiskt brus (eng. *isotropic noise*) eller anisotropiskt brus (eng. *anisotropic noise*). För isotropiskt brus innebär det att bruset som genereras visuellt inte kan definieras med en enskild riktning medan anisotropiskt brus har en tydlig visuell riktning (figur 2). Mönstret som kommer av riktningen i anisotropiskt brus kan likna bland annat det av fiberriktningen i trä. Vid implementering av isotropiskt eller anisotropiskt brus innebär det generellt ett övervägande för när dessa är lämpliga i sammanhanget. Diverse material kan presentera samma visuella riktningsegenskap och vid tillfällen när dessa material ska genereras bör denna egenskap överensstämma med det efterliknade materialets (Pasbanigoloojeh, Lawal, Daneshvar & Lawal 2026).



Figur 2: Höjdkartan till vänster visar isotropiskt brus av Simplex Noise av Peck (2016) medan den till höger visar anisotropiskt brus av Gabor Noise av Jijup (2020)

Vid användandet av brusalgoritmer är det vanligt att tillämpa *Fractal Noise*. Det är en teknik baserat på att sammanställa flera överlappande lager av brus, även kallat oktaver (Emmanuel et al. 2019). Dessa överlappande lager kan variera, främst beroende på vilken brusalgoritm som används men även utifrån den angivna viktfaktorn. Viktfaktorn i lagren påverkar sedan lagrets betydelse när det sammanställs till *Fractal Noise*. Tekniken används främst för att skapa en högre variation av detaljer i det genererade bruset (Galín et al. 2019; Zhang et al. 2025).



Figur 3: Visualisering av de olika bruslagren för Fractal Noise. Bruset är genererat från Simplex Noise av Peck (2016)

2.5 Lattice Gradient Noise & Sparse Convolution Noise

Lagae et al. (2010) presenterar kategoriseringar av brusalgoritmer utifrån egenskaper som särskiljer dem åt. Till dessa olikheter inkluderar Lagae et al. (2010) även algoritmernas lämplighet för att tillämpas inom olika områden. Brusalgoritmerna behöver inte begränsas till endast en kategori utan kan också överlappa mellan flera kategorier. Två av dessa kategorier är *Lattice Gradient Noise* och *Sparse Convolution Noise*.

Kategorier särskiljs bland annat genom de tillvägagångssätt som respektive algoritmer använder för att generera brus. *Lattice Gradient Noise* producerar brus genom att skapa rutnät, där varje nod sedan används för att blanda gradienter eller värden. Algoritmer tillhörande denna kategori är bland annat *Perlin Noise*, *Simplex Noise* och *Flow Noise*. *Sparse Convolution Noise* sammanställer effekten av flera viktade punkter och deras tillhörande effekt baserat på slumpad placering. Denna effekt avtar gradvis beroende på avståndet till dess mittpunkt. *Gabor Noise* och *Spot Noise* tillhör denna kategori av brus.

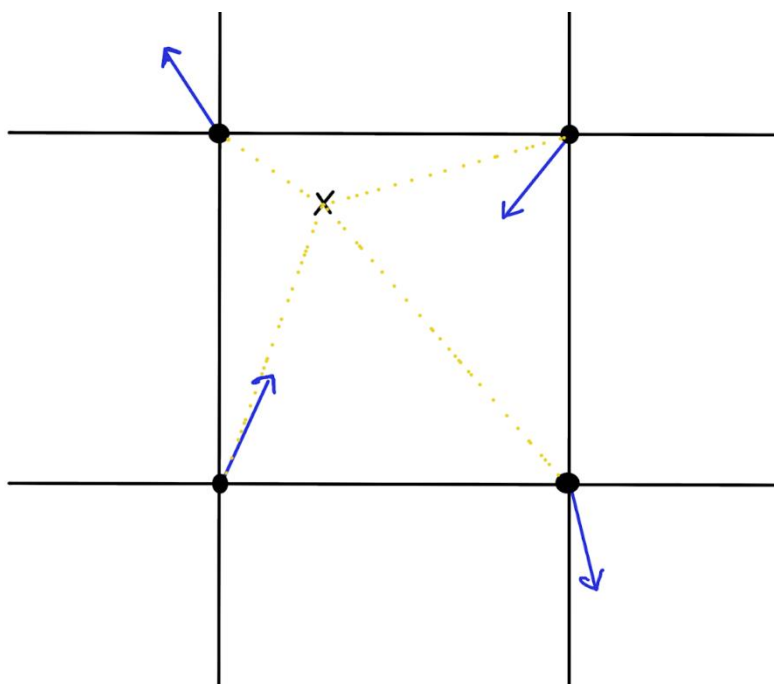
2.6 Perlin

I artikeln *An Image Synthesizer* (1985) presenterar Ken Perlin en metod för att generera sammanhängande och gradientbaserat brus. Denna metod kommer senare att utgöra grunden för vad som slutligen blir benämnt *Perlin Noise*. I ett senare arbete av Ken Perlin (2002) vidareutvecklar han algoritmen vilket sedan resulterar i färre visuella artefakter och förbättrad exekveringstid vid generering av *Perlin Noise*.

2.6.1 Algoritmen

Perlin Noise tillhör den tidigare beskrivna kategorin *Lattice Gradient Noise* och följer övergripande samma tillvägagångssätt som kategorin beskriver hur brus genereras

(Lagae et al. 2010). För att beräkna en pixel, beskriver Perlin (2002) mer detaljerat kring hur metoden genomförs. *Perlin Noise* konstruerar ett rutnät som upprepas regelbundet och i godtycklig dimension. För varje intilliggande nod anges en pseudoslumpad gradientvektor via en hashfunktion, som baseras på fördefinierade värden. Nästa steg i processen är att beräkna vektorerna mellan positionen på pixeln och relevanta nodpunkter. För varje vektor och tillhörande gradientvektor beräknas sedan skalärprodukten. Slutligen genomförs bilinjär interpolering för att erhålla mjuka övergångar i resultat, som är baserat på skalärprodukternas påverkan i förhållande till pixelns placering inom rutnätet.



Figur 4: Visualiseringen av delsteg inom Perlin Noise. Delstegen som representeras är rutindelning, tillhörande noder med gradientvektorer och markeringar för vektorerna mellan pixeln och noderna.

Gällande tidskomplexiteten presenterar Perlin (2001) följande resonemang. Antalet noder som *Perlin Noise* använder växer exponentiellt med dimensionen N och resulterar i en komplexitet på $O(2^N)$. Vidare är diverse vektorberäkningar kopplat till dimensionen N som brukas och tillför en komplexitet av $O(N)$. Den sammanställda tidskomplexiteten är grundad i dimensionen som rutnätet och vektorberäkningarna utförs i, vilket därmed blir tidskomplexiteten:

$$O(N \times 2^N) \quad (1)$$

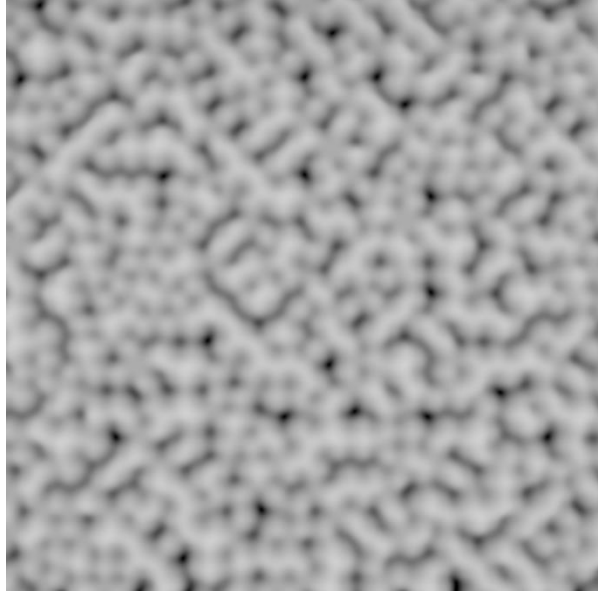
Perlin Noise har alltså en exekveringstid som är konstant per dimension. Experimentet i detta arbete genomförs endast i 2D vilket därmed blir följande konstanta komplexitet per pixel:

$$O(2 \times 2^2) \quad (2)$$

2.6.2 Visuellt karaktär

Formellt beskrivs *Perlin Noise* som isotropiskt brus med mjuka övergångar och en 2D-höjdkarta av algoritmen visas i figur 5. Trots vidareutvecklingen presenteras fortfarande

visuella artefakter i bruset. Zheng et al. (2025) beskriver några artefakter som nätliknande mönster och är potentiellt det som Perlin (2001) och Lagae, Lefebvre och Dutré (2011) anser vara anisotropiska artefakter inom *Perlin Noise*. Slutligen framför även Wang et al. (2025) att vid högre dimensioner exponeras artefakterna mer tydligt.



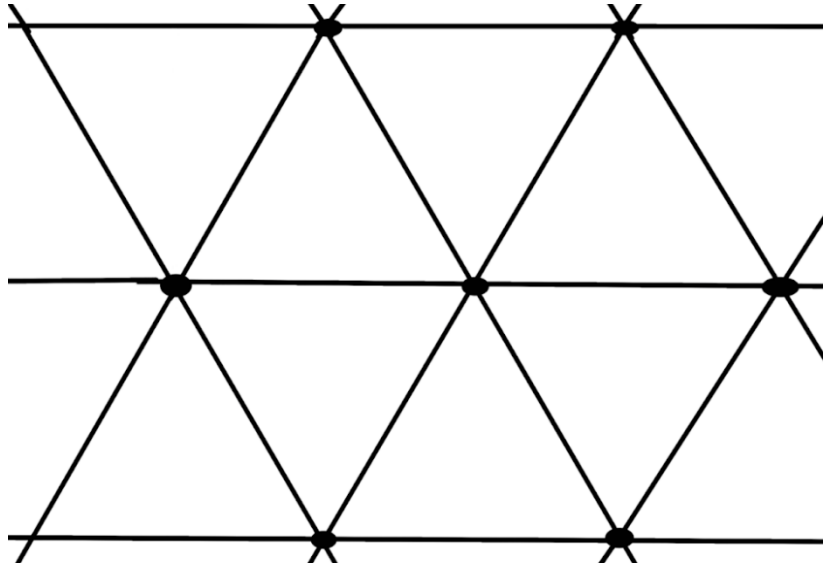
Figur 5: 2D-isotropiskt Perlin Noise av brusimplementationen från Takahashi (2015)

2.7 Simplex

Algoritmen beskrivs först av Perlin (2001) och presenteras som en förbättring av *Perlin Noise*. Primärt vid hanteringen av högre dimensioner men även sekundärt i minskandet av mängden visuella artefakter i slutresultatet (Perlin 2001; Zhang et al. 2025).

2.7.1 Algoritmen

Simplex Noise är övergripande lik *Perlin Noise*; båda algoritmerna tillhör *Lattice Gradient Noise* och genomför liknande tillvägagångssätt som kategorin beskriver (Lagae et al. 2010). Värt att notera är att algoritmen fortfarande implementerar ett tillvägagångssätt distinkt från *Perlin Noise*. Den främsta skillnaden förekommer i användandet av ett kontinuerligt simplicialt nät (eng. *simplicial grid*) istället för ett kontinuerligt rutnät (Perlin 2001). Det simpliciala nätet består av flera simplexer, som definieras utifrån dimension för implementationen och antalet noder ökar linjärt utifrån dimensionen (N) som följande: $N+1$. Inom 2D antar simplexer den geometriska formen av liksidiga trianglar (Boyd & Vandenberghe 2009, s. 33). Se figur 6 för en visualisering av detta 2D-nät bestående av simplexer. Till detta nät används också en funktion som avtar symmetriskt i samtliga riktningar utifrån en nod, varav *Perlin Noise* istället brukar bilinjär interpolering mellan noder.



Figur 6: Visualisering av ett simplex-nät i 2D bestående av trianglar.

Trots att algoritmerna har många likheter i sitt utförande har *Simplex Noise* en teoretiskt lägre tidskomplexitet. I likhet med *Perlin Noise* är implementationen påverkad av dimension N som den används inom. Perlin (2001) resonerar att tidskomplexiteten grundas i det simpliciala nätet och att dess antal noder samt användningen av vektoroperationer ökar linjärt med dimensionen; $O(N)$. Enligt Perlin (2001) har *Simplex Noise* därmed tidskomplexiteten:

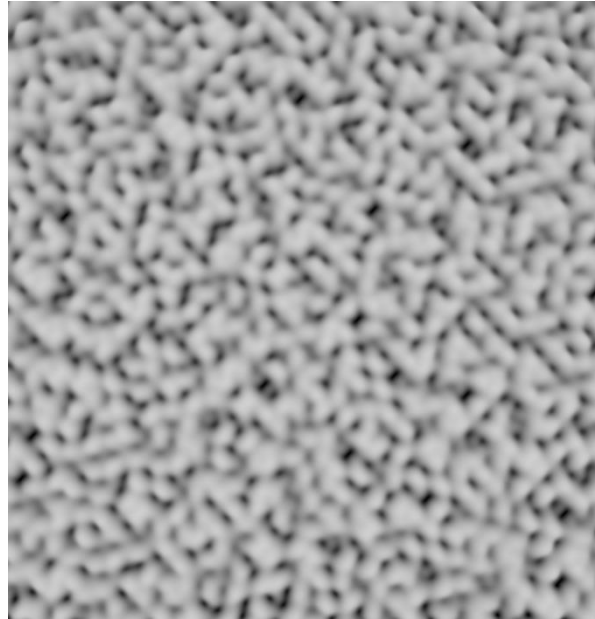
$$O(N^2) \quad (3)$$

Både Perlin (2001) och Zheng et al. (2025) framför att *Simplex Noise* jämfört med *Perlin Noise* har lägre tilläggs-kostnader på högre dimensioner, men detta arbete utför endast 2D-*Simplex Noise* med konstant tidskomplexitet:

$$O(2^2) \quad (4)$$

2.7.2 Visuell karaktär

Bruset som genereras av *Simplex Noise* är, i likhet med *Perlin Noise*, isotropiskt brus och saknar därmed visuell riktning. Visuellt framstår algoritmen som mer enhetlig än dess föregångare *Perlin Noise* med färre påträffade artefakter av anisotropisk karaktär (Perlin 2001). Ett exempel på brus från en implementation av *Simplex Noise* finns i figur 7 nedanför.



Figur 7: 2D-isotropiskt Simplex Noise av brusimplementationen från Peck (2016)

Diverse visuella artefakter som framkommer i *Perlin Noise* har huvudsakligen motverkats av förändringarna i tillvägagångssättet för *Simplex Noise*. Gustavson och McEwan (2022) presenterar dock ett fynd av att visuella artefakter fortfarande existerar fast inom ett annat plan, vilket liknar artefakterna som uppstår i *Perlin Noise*.

2.8 Gabor

Lagae, Lefebvre, Drettakis och Dutré (2009) presenterar en brusalgorithm som bygger vidare på algoritmen *Sparse Convolution Noise* genom att introducera en *Gabor*-kärna (eng. *gabor kernel*). Utvecklingen av algoritmen ämnar att kunna generera isotropiskt som anisotropiskt brus men också för att förbättra kvaliteten på anisotropisk filtrering på bruset.

2.8.1 Algoritm

Gabor-kärnan fungerar som en extra parameter i algoritmen och introducerar ett område som påverkas av kärnans periodiska effekt. Effekten är som starkast i mittpunkten och avtar beroende på avståndet till sagd punkt. *Gabor*-kärnan består av en cirkulär Gaussisk funktion som är multiplicerad med en sinusvåg. När bruset genereras så fördelas planet in i ett rutnät för att deterministiskt sprida *Gabor*-kärnor med *poissonfördelning*. Fördelningen sker inom cellen som pixeln tillhör och angränsande celler. Vad som anses vara angränsande celler förtydligas inte ytterligare. Det rekommenderas att varje cell ska vara lika stor som *Gabor*-kärnans radie. Slutligen summeras *Gabor*-kärnornas påverkan för den aktuella pixeln (Lagae et al. 2009).

För att kunna verkställa en fullvärdig exekvering av algoritmen krävs parametrar som anger värden för *amplitud*, *frekvens*, *kärnbredd*, *impulser per kärna* och *riktning*. *Amplituden* påverkar styrkan på det genererade bruset medan *frekvensen* påverkar tätheten av bruset. *Kärnbredden* påverkar hur snabbt effekten avtar baserat på distansen från kärnorna och antalet impulser påverkar fördelningen av *Gabor*-kärnorna. Parametern för *riktning* avgör om bruset ska vara isotropiskt eller anisotropiskt med

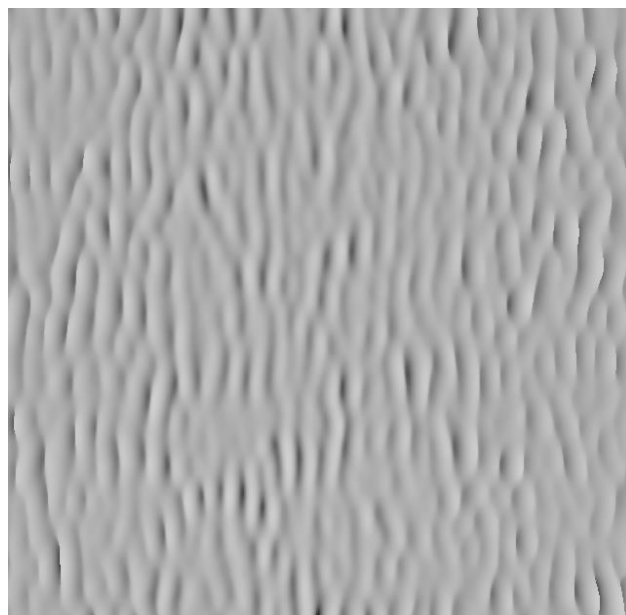
bestämd riktning. *Gabor Noise* är inte den enda brusimplementationen som har *amplitud* och *frekvens* utan dessa används även i *Perlin Noise* och *Simplex Noise*.

Gällande *Gabor Noise* kunde ingen relevant källa hittas som konkret analyserar och beskriver tidskomplexiteten för algoritmen, men utifrån beskrivning av algoritmen från Lagae et al. (2009) kan en eventuell tidskomplexitet härledas. För varje pixel som beräknas fördelas ett fast antal kärnor inom varje enskild cell för samtliga celler, vilket kan gälla ett 3×3 -rutnät i 2D eller $3 \times 3 \times 3$ -nät i 3D. Om följande utveckling av rutor gäller i dimensionen N kan tidskomplexiteten möjligen beskrivas med följande konstanta uttryck per dimension:

$$O(3^N) \quad (5)$$

2.8.2 Visuell karaktär

Jämfört med *Simplex Noise* och *Perlin Noise* finns möjligheten att generera isotropiskt och anisotropiskt brus med *Gabor*-algoritmen. Detta innebär att en anisotropisk implementation i en 3D-miljö kan konstrueras för att bibehålla detaljnivåer på längre distanser. Nedanför i figur 8 kan den anisotropiska effekten av *Gabor Noise* observeras på bruset. Utöver det reflekterar kvaliteten på bruset utifrån prestandakraven som fastställs för algoritmens parametrar. Låg prestandapåverkan innebär sämre kvalitet medan tyngre implementationer resulterar i förhållandevis bättre kvalitet (Lagae et al. 2009).



Figur 8: 2D-anisotropiskt *Gabor Noise* av brusimplementationen från Jijup (2020)

2.9 Tidigare arbeten

Tidigare arbeten som berör frågeställningen i denna studie är bland annat arbetet av Lagae, Lewis och Dutré (2011) som delvis innehåller en jämförelse av exekveringstider på GPU:n utifrån en upplösning mellan *Perlin Noise* och den förbättrade *Gabor Noise*. Resultatet presenteras i FPS, där *Perlin Noise* resulterar i avsevärt högre FPS än *Gabor Noise* och därmed betydligt snabbare exekveringstid.

Vitacion och Liu (2019) utför en prestandaanalys av *Perlin Noise* och *Simplex Noise* där algoritmernas exekveringstider och minnesanvändning är i fokus. I deras arbete lagras mätdata från flera olika hårdvaror, där fokus ligger på CPU:ns exekveringstid. Deras resultat visar övergripande att *Simplex Noise* har snabbare exekveringstid än *Perlin Noise*.

Slutligen presenteras ett kandidatarbete av Strand och Prestberg (2025) som delar många likheter med detta arbete, främst inom syfte, metodik och resultat. Studien jämför exekveringstiden mellan olika externa implementationer för brusalgoritmerna *Perlin Noise* och *Simplex Noise*. I deras resultat finns stora skillnader i exekveringstiden mellan implementationerna av samma algoritm. De redogör att den genomsnittliga exekveringstiden för *Simplex Noise* är snabbare än *Perlin Noise* trots variationen mellan implementationerna.

3 Problemformulering

Den valda frågeställningen som utforskas i arbetet är följande: “Hur presterar olika brusimplementationer i realtidstillämpningar utifrån varierande upplösningar?”

Inom spel finns ett behov av att bibehålla en relativt låg exekveringstid och jämn FPS för att undvika att påverka spelarens upplevelse negativt (Liu et al. 2023). För spel med realtidsbaserad generering är detta av ytterligare betydelse för att stora delar av spelvärlden kan behöva genereras, utan att negativt påverka spelupplevelsen.

Studien ämnar att primärt besvara frågeställningen men också bidra med faktiska exekveringstider som är direkt beroende av hårdvaran och kan möjligen visa hur lämpliga implementationerna är inom realtidsanvändning. Vidare kan den insamlade informationen från flera brusimplementationer möjligen bidra till en jämförelse mellan algoritmerna.

Detta arbete inkluderade en uppsättning algoritmer som tidigare arbeten inte har studerat gentemot varandra, samt att antalet algoritmer är fler än i bland annat arbetet från Strand och Prestberg (2025). Vidare användes bland annat flera implementationer per brusalgoritm, samt att många tidigare arbeten inkluderar endast ett fåtal upplösningar varav denna studie har fler och mer närliggande upplösningar.

3.1 Hypotes

Inför arbetet har några hypoteser konstruerats som bedöms kunna bli bekräftade av resultaten från studien. Som tidigare beskrivits så presenteras *Simplex Noise* som en förbättrad variant av *Perlin Noise* när det gäller hur exekveringstiden utvecklas beroende på den avsedda dimensionen (Perlin 2001; Zhang et al. 2025). Vidare utforskades endast tvådimensionella brusimplementationer i detta arbete, vilket innebar att fördelen som *Simplex Noise* innehar vid beräkningar med högre dimensioner inte bidrog avsevärt till resultatet.

Jämfört med resterande algoritmer genomför *Gabor Noise* en mer komplex process som tidigare har beskrivits. Parametrarna för *Gabor Noise* har en betydande roll och bör övervägas beroende på om genereringen främst ska utföras snabbt eller med bättre kvalitet (Lagae et al. 2009; Lagae et al. 2010).

Utifrån dessa grunder presenteras en hypotes om att implementationer med *Perlin Noise* och *Simplex Noise* generellt presterar med liknande resultat. Vad gäller implementationer av *Gabor Noise* kommer dess exekveringstid potentiellt bli måttligt till betydligt högre jämfört med resterande brusalgoritmer.

3.2 Metodbeskrivning

För datainsamlingen genomfördes ett experiment i en egenkonstruerad testmiljö. I experimentet producerar de olika brusimplementationerna brus med endast en oktav. Genereringen av brus sker utifrån höjdkartor med olika upplösningar, där upplösningen representerar en problemstorlek i experimentet. Av experimentet erhålls kvantitativa rådata i form av exekveringstid i millisekunder (ms), som mäts av den inbyggda C#-klassen *Stopwatch* (Microsoft 2026). Experimentet begränsas till att endast utföra beräkningarna sekventiellt på CPU:n och datainsamlingen sker på två datorer med olika

hårdvaror.

Varierande upplösningar för höjdkartan används för en bredare datainsamling. Problemstorlekarna följer en kvadratisk sekvens som utvidgas successivt, som börjar vid 1×1 och avslutas när den övre gränsen för problemstorleken har genomförts. Denna övre gräns varierar mellan algoritmerna; *Gabor Noise* har en gräns på 256×256 medan *Simplex Noise* och *Perlin Noise* har en gräns på 768×768 . För varje problemstorlek genomförs sedan 100 iterationer, varav varje iteration består av en tidtagning. Vidare anges ett slumpat *seed*-värde per iteration som sedan påverkar utseendet för resulterande brus från implementationerna. Detta *seed* överensstämmer i samtliga tester av implementationerna och varierar endast mellan iterationerna.

Ur rådatan kommer värden som lägsta och högsta exekveringstiderna att identifieras, samt förväntas datamängden kunna brukas för beräkningar av median per upplösning. Experimentet kommer att utvärdera tre brusalgoritmer med högst två olika implementationer för vardera algoritm. De som inkluderas är följande:

- *Perlin Noise*: Peck (2016) och Takahashi (2015)
- *Simplex Noise*: Peck (2016) och Rombauts (2014)
- *Gabor Noise*: Jijup (2020)

3.3 Metoddiskussion & problematisering

3.3.1 Anpassningar och val

Ett tillvägagångssätt som användes för att förbättra resultatet var flera iterationer. Varje implementation körde sina problemstorlekar i 100 iterationer för att få mer tillförlitliga resultat, vilket verkar vara någorlunda vanligt för att etablera en tillförlitlig mängd data (Vitacion & Liu 2019; Beiranvand, Hare & Lucet 2017).

Valet av att inkludera exekveringstiden i rådatan valideras av Beiranvand, Hare och Lucet (2017), som beskriver parametern som ett vanligt mätvärde för att definiera algoritmers prestanda. Vidare implementerar flera studier datainsamlingar bestående av exekveringstiden för att kartlägga och jämföra algoritmer (Lagae, Lewis & Dutré 2011; Strand & Prestberg 2025; Vitacion & Liu 2019). Med tillgång till verktyg som *Stopwatch*-klassen (Microsoft 2026) för hantering av tidtagningen på CPU:n, bidrar detta med att underlätta genomförandet av insamlingen av exekveringstiden. Samtliga delar av detta ligger till grund för varför detta arbete inkluderat exekveringstiden som mått för prestandan för brusalgoritmerna.

Det finns kompletterande alternativ till exekveringstiden för att mäta hur algoritmer presterar, varav Beiranvand, Hare och Lucet (2017) bland annat nämner minnesanvändning. Vitacion och Liu (2019) har minnesanvändningen som en del av metoden främst för att erhålla data för minnesanvändning som brukas vid skapandet av höjdkartor. Att inkludera minnesanvändningen för algoritmer bidrar till en djupare förståelse men det utvidgar omfånget på arbetet och därav exkluderades minnesanvändningen från detta arbete.

Andra arbeten inkluderar flertalet oktaver i experimenten (Vitacion & Liu 2019), vilket konsekvent resulterar i större skillnader i exekveringstider samt mer detaljerat brus.

Detta arbetes experiment inkluderar endast en oktav och grundar detta i potentialen med att använda stigande upplösningar för högre exekveringstider, samt för att redovisa en mer isolerad jämförelse som inte inkluderar brustekniker som *Fractal Noise*.

Från tidigare arbeten har eventuella brister konstaterats, som detta experiment försöker att undvika. Flera verk genomförde datainsamlingar som inte inkluderar primärdata utan endast bearbetad data, som medelvärdet från mätningarna (Strand & Prestberg 2025; Vitacion & Liu 2019). Detta innebär att tillvägagångssättet möjligtvis döljer diverse data från att analyseras och därmed påverkar trovärdigheten i resultatet. För att öka tillförlitligheten ytterligare bearbetas sedan rådatan utifrån värden för minimum, median och maximum som efterliknar delar av arbetet från Silva, Arcaro och de Oliveira (2020).

Valet att inkludera två olika datorer i experimentet grundas i att kunna producera data som har en större bredd och som är mindre riktad gentemot enskild hårdvara. Vitacion och Liu (2019) genomför datainsamling med flera hårdvaror i deras prestandajämförelse av flera brusalgoritmer. Fördelen med detta är att förhoppningsvis kunna redovisa eventuella prestandaskillnader utifrån olika mönster från hårdvaran, samt förbättra möjligheten att ge bättre belägg för vilka sammanhang som resultaten gäller.

3.3.2 Upplösningar

Flera tidigare arbeten som utfört liknande experiment saknar en successiv upptrappning som täcker många upplösningar. Utifrån detta utvecklades metoden för att samla in rådata med problemstorlekar som bidrar till en mer fulländad genomgång av olika upplösningar, som också i ett senare skede kan bearbetas.

För att erhålla trovärdigare resultat implementeras en stor variation av upplösningar som problemstorlekar. Detta grundas i att högre upplösningar innebär ökade exekveringstider från algoritmerna och därmed tydliggör tidsskillnader. Beiranvand, Hare och Lucet (2017) kompletterar detta resonemang med att beskriva hur problemstorleken ska vara tillräckligt bred för att kunna bidra med mer heltäckande information kring varje algoritm.

Värdet för upplösningarna härstammar från att implementering av brus vanligtvis appliceras in i en kvadratisk höjdkarta. I och med detta föreligger förhoppningar om att kunna utvinna data relevant för vilka upplösningar som brusalgoritmer används för i praktiken. Den främsta bristen med detta tillvägagångssätt är att problemstorlekarna växer exponentiellt och bildar fortfarande luckor av problemstorlekar som inte hanteras av experimentet.

Pilottester genomfördes bland annat för att se hur lång tid datainsamlingen tog för implementationerna. När detta prövades upptäcktes det att *Gabor Noise* hade avsevärt högre beräkningstider redan vid låga upplösningar. På grund av detta begränsades upplösningen till maximalt 256×256 för *Gabor Noise*, medan resterande implementationer använde 768×768 . Dåvarande preliminära resultat ansågs fortfarande vara applicerbara för realtidsbaserade lösningar.

3.3.3 Valet av CPU-implementation

Den huvudsakligen funna forskningen relaterar främst till brusalgoritmers

exekveringstid i sammanhanget av GPU-implementation, som i arbetet av Gustavson och McEwan (2022). Vidare så framförs behovet av anpassningar på brusalgoritmerna för att kunna exekvera beräkningarna på GPU:n (Gustavson & McEwan 2022) eller för att GPU:n inte ska bli en flaskhals (eng. *bottleneck*) och resultera i orimliga exekveringstider (Olano 2005). Utifrån kunskapsluckan som uppkommer av att forskningen främst prioriterar exekveringstider från GPU:n och att algoritmer potentiellt inte är utvecklade med GPU:ns arkitektur i beaktning, genomförs datainsamlingen endast på CPU:n.

I sammanhanget av att det anpassas för CPU:n sker experimentet sekventiellt och undviker att utforska parallellisering. Åtgärden vidtas främst för att minimera potentiella omkostnader (eng. *overhead*) som är extra beräkningskostnader oberoende av brusimplementationerna. Tillfällen då omkostnader kan förekomma är vid till exempel introduceringen av flera trådar, *multithreading* (Casini 2022). Ytterligare motiveras valet av att begränsa storleken på arbetet till en mer hanterbar arbetsmängd.

3.3.4 Val av algoritmer och avgränsningar

Valet av algoritmer genomfördes metodiskt, *Perlin Noise* är en allmänt känd brusalgoritm medan *Simplex Noise* anses ha snabbare exekveringstid än *Perlin Noise* (Zheng et al. 2025) men producerar liknande resultat. Båda dessa kan kategoriseras som *Lattice Gradient Noise* enligt Lagae et al. (2010). I relation till detta valdes *Gabor Noise* främst för att den tillhör en annan kategori, nämligen *Sparse Convolution Noise*. Lagae et al. (2010) beskriver denna kategori som generellt mer beräkningstung men samtidigt med mer detaljerade resultat. En annan fördel med att inkludera *Gabor Noise* är möjligheten till att kunna utvärdera skillnaderna i tillvägagångssättet i hur algoritmen producerar brus jämfört med *Lattice Gradient Noise*-algoritmerna.

Olika brusalgoritmer är utvecklade utifrån olika ändamål och sammanhang, detta innebär sedan variationer i tillvägagångssättet för att beräkna och konstruera respektive brus. *Gabor Noise* kan till exempel vara långsammare jämfört med andra brusalgoritmer men möjliggör mer organiska och realistiska resultat (Lagae et al. 2010). Ett annat exempel är att *Simplex Noise* jämfört med *Perlin Noise* presterar bättre i högre dimensioner. Utifrån denna beskrivning kan det finnas ett värde i kunskapen om för vilka sammanhang som algoritmerna är lämpliga att användas. Till detta kan alltså exekveringstiden bli relevant beroende på hur brusalgoritmen ska användas.

En avgränsning som gjordes var att utesluta flera olika uppsättningar av parametrar för *Gabor Noise*. För att inkludera flera olika uppsättningar av parametrar hade det genererade bruset behövt kontrolleras noggrant. Kontrollen skulle bestå av att manuellt avgöra när bruset övergår till att vara estetiskt icke-kontinuerligt, vilket enligt Lagae et al. (2010) inte anses vara fullvärdigt procedurrellt brus. Vidare ska hanteringen av data för samtliga uppsättningar parametrar presenteras och analyseras. Därav anses tillhörande prövning av olika uppsättningar av parametrar för resurskrävande.

Inkluderingen av dubbla implementationer beror på att algoritmer kan tolkas annorlunda inför en implementering. Därmed inkluderas fler implementationer för att förtydliga eventuella skillnader som kommer från den enskilda implementeringen av samma algoritm. Detta i kombination med att de inkluderade algoritmerna är en liten delmängd av den totala mängden brusalgoritmer innebär mindre övergripande prestationsjämförelser gentemot utbudet av alla brusalgoritmer. Däremot är målet med

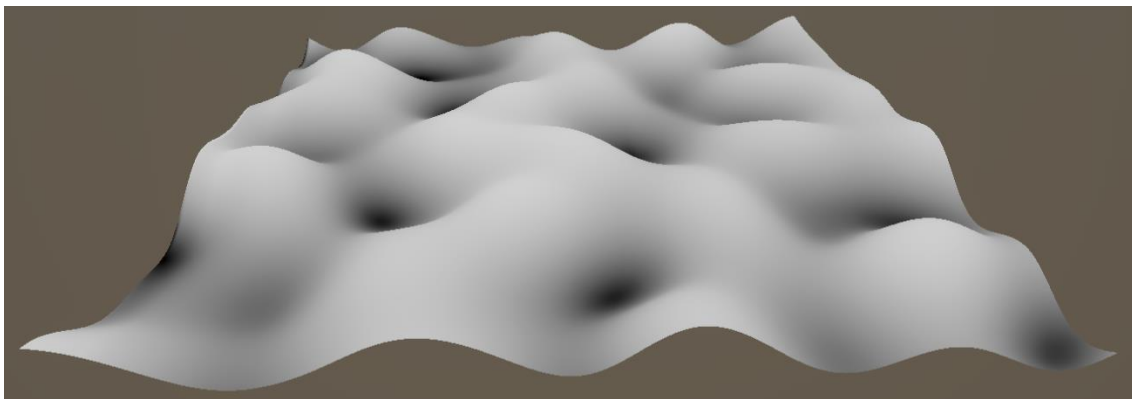
flera implementationer att kunna bidra positivt till undersökningen genom att göra den mer nyanserad för varje algoritm som inkluderats.

4 Genomförande

Utmed arbetet genomfördes ett praktiskt arbete i form av skapandet av en artefakt med samtliga brusimplementationer. Artefaktens syfte var att samla in exekveringstiden och säkerställa implementationernas korrekthet. Resultatet analyseras sedan för att se om det går att avgöra vilken implementerad algoritm som är bäst lämpad för realtidsbaserad användning.

4.1 Hanteringen av brusimplementationer

I det tillgängliga urvalet av implementationer var prioriteringen främst utvecklare med erfarenhet inom programmering. Vidare validering skedde också med manuell genomgång för att försöka säkerställa att implementationen har en tydlig koppling till den tillhörande algoritmen. Det genomfördes en visuell validering av implementationerna med en komponent för att kontrollera att generering av brus var enligt förväntan utifrån algoritmerna. Visualiseringen sker genom att höjdkartor konstrueras med olika upplösningar för att sedan användas för att generera en 3D-mesh som visualiserar bruset från implementationen (figur 9). Dessa höjdkartor jämförs sedan med de ursprungliga algoritmerna för att försöka verifiera implementationen.



Figur 9: Resultande visualisering med Simplex Noise av Peck (2016) som lågfrekvent brus.

Samtliga algoritmer ämnar att ha två implementationer men eftersom utbudet av *Gabor Noise* var begränsat och ingen ytterligare implementering än den av Jijup (2020) kunde inkluderas. Detta berodde på att ingen annan implementering kunde verifieras som korrekt eller tillräckligt distinkt från den av Jijup (2020) och därmed presenteras endast en implementering för *Gabor Noise*. Det har övervägts att byta ut algoritmen mot andra mer tillgängliga brusalgoritmer. På grund av att Lagae et al. (2010) beskriver kategorierna *lattice gradient noise* och *sparse convolution noise* utifrån inkluderade algoritmer bedömdes kategoriseringen som eventuellt underlag för framtida jämförelser.

Samtliga brusimplentationer har hämtats från externa källor och har därmed inte ursprungligen utvecklats med samma programmeringsspråk som testmiljön (Jijup 2020; Rombauts 2014; Zhang 2018). Detta innebar att diverse korrigeringar genomfördes för att konvertera implementationen till C# med fokus på att efterlikna den ursprungliga brusimplementationen. Därmed prioriteras motsvarigheter inom programmeringsspråket före varianter från Unity. Detta gäller även metoder från den ursprungliga lösningen som utvecklats separat för specifika ändamål inom algoritmen

men som redan har färdigställda klasser och metoder inom C#, såsom *System.Random* för pseudoslumpade värden. I detta fall översätts de ursprungliga metoderna till C# för att minimera skillnader.

Förändringar och val har genomförts inom implementationerna för att endast inkludera det som är relevant för experimentet. Exempelvis inkluderade hela koden för *Gabor Noise* av Jijup (2020) hanteringen av anisotropiskt och isotropiskt brus varav den korrigerade implementationen endast inkluderar anisotropisk brusgenerering. Liknande korrigering gäller för hantering av vilken interpolering *Perlin Noise* av Peck (2016) brukar. I slutändan valdes den tyngre femtegradspolynoma interpoleringen (eng. *quintic interpolation*). Vidare för implementationerna av *Perlin Noise* och *Simplex Noise* från Peck (2016) så tillgodoser utvecklaren möjligheten att välja mellan 2 olika precisioner; 32-bitar och 64-bitar. För dessa implementationer används 32-bitars precision i experimentet.

Gabor Noise särskiljer sig i förhållande till resterande brusalgoritmer i miljön för dess krav på ytterligare parametrar som inte förekommer i *Perlin Noise* eller *Simplex Noise*. Vidare har implementationen av Jijup (2020) ingen parameter för impulsdensitet (λ) som Lagae et al. (2009) har utan beräknar detta värde utifrån parametern för kärnbredden (a). Valet av värden för parametrarna grundas i att de genererade resultaten inte ska vara beräkningstunga samtidigt som det uppnår visuellt acceptabla nivåer (Lagae, Lewis & Dutré 2011). Med detta i åtanke testades olika parametrar med verktyget som visualiserar brus för att avgöra passande värden till *Gabor*-implementationen. Värdena som slutligen valdes för parametrarna presenteras i tabell 1.

Parameter	Värde
Amplitud (K)	2
Kärnbredd	0,25
Frekvens (F_0)	0,35
Impulser per kärna	32

Tabell 1: Parametrar och respektive värde för implementationen av *Gabor Noise*.

4.2 Artefakt

I denna studie har en artefakt konstruerats med syftet att möjliggöra automatiserad datainsamling och utvecklades inom C# och Unity (version 6000.0.58f2). Den främsta anledningen till att C# och Unity valdes för att framställa artefakten beror på att författarna har tidigare erfarenhet inom dessa. Unity har även rendering inbyggt, vilket underlättar processen för att genomföra valideringen av brusimplementationerna. Användandet av Unity finner även ett syfte i att brus är vanligt inom spel och eventuellt finns det då relevans i att resultatet kommer från en spelmotor såsom Unity. All kod som har implementerats för experimentet i Unity har begränsats till att endast utföras sekventiellt på CPU:n som beskrivs i studiens metod.

Burst Compiler (Unity Technologies 2026a) är ett optimeringsverktyg som är tillgängligt

i Unity. För att kunna använda detta verktyg som används för att öka prestandan inom implementeringarna behöver koden korrigeras för att bli *burst*-kompatibel. Därmed introduceras inte *Burst Compile* i implementeringarna, för att förhindra påtagliga förändringar gentemot den ursprungliga koden.

Vidare består artefakten av en testmiljö strukturerad för att kunna samla in exekveringstider från samtliga implementationer. Samtidigt som den strävar efter att upprätthålla så identiska förhållanden som möjligt. Detta genomförs genom att använda samma omkringliggande kodstruktur för att beräkna pixelns position i bruset för samtliga implementationer.

Tidtagningen sker per iteration och inkluderar endast brusimplementationens exekvering för en enskild problemstorlek. Detta innebär att diverse tidskostnader som tillkommer av exempelvis rendering och händelser (eng. *events*) inte inkluderas. Utöver det används en kompilerad och optimerad version av applikationen för att minimera fluktuationer som potentiellt tillkommer av Unitys utvecklingsmiljö (Unity Technologies 2026b). Detta görs för att minimera faktorer som skulle kunna leda till missvisande resultat.

Efter varje iteration lagras exekveringstiden i en klass utvecklad för datahantering. Det huvudsakliga syftet med denna klass är att strukturera datan för att sedan konstruera en CSV-fil (eng. *comma separated value file*) efter att den sista problemstorleken har genomförts.

4.2.1 Stopwatch

Ett problem som framförs av Akinshin (2019, s.637) är tidtagning med *Stopwatch-klassen* (Microsoft 2026). Författaren presenterar diverse data rörande tillförlitligheten i tidtagningen och förmedlar att det inte är ett perfekt verktyg för tidtagning. Detta kan leda till felaktigheter i rådatan som eventuellt kan påverka resultatets trovärdighet. Utifrån detta har urvalet av verktyg för tidtagning setts över för att eventuellt finna ett mer noggrant verktyg eller för att validera en fortsatt användning av *Stopwatch-klassen* (Microsoft 2026).

Ett alternativ till *Stopwatch* är profileringsverktyget *Unity Profiler* (Unity Technologies 2025) som har en bred kapacitet för att samla in en mängd olika parametrar som anses relevanta för att avgöra prestanda. Det förekommer arbeten som genomför experimentet med *Unity Profiler* men sällan endast för exekveringstiden på CPU:n (Koulaxidis & Xinogalos 2022; Marin et al. 2025). I detta experiment anses endast exekveringstiden på CPU:n relevant, vilket betyder att stora delar av verktyget behöver exkluderas korrekt. Vidare kunde ingen källa hittas som presenterar eventuella brister i träffsäkerheten med profileringsverktyget. Ett annat alternativ som övervägdes var Unitys *Time*-klass och specifikt funktionen *Time.realtimeSinceStartupAsDouble* (2026c). I den tillhörande dokumentationen så förmedlar utvecklarna däremot en viss osäkerhet kring funktionens lämplighet för tidtagning inom samma bildruta (eng. *frame*) på grund av att tidsdifferensen kan resultera i noll vilket ledde till uteslutandet av detta alternativ.

Trots att *Stopwatch-klassen* (Microsoft 2026) har brister i träffsäkerheten handlar det om försumbart korta nanosekunder. Jämfört med *Unity Profiler* så bidrar vetskapen om träffsäkerheten med klargörelse och mindre spekulationer kring tidmätningens påverkan. Trots bristen beskriver Akinshin (2019, s.633) *Stopwatch* som positivt och

förmedlar att verktyget är det främsta inom *.Net*. Ytterligare bör det framföras att *Stopwatch* har brukats tidigare inom andra experiment för tidtagning av exekveringstider och är därmed inget ovanligt verktyg för detta ändamål (Odermatt, Marcilio & Furia 2022; Dyrda et al. 2022). Ytterligare, med hjälp av funktionen *IsHighResolution* kan testmiljön säkerställa en mer träffsäker implementation som använder en *high-resolution performance counter* istället för systemets tidtagning (Microsoft 2026). Slutligen anses möjligheten att kunna minska risken för påverkan från den mänskliga faktorn med en implementering av *Stopwatch* som fördelaktig.

4.3 Insamling och hantering av data

Inför datainsamling genomfördes ett pilottest för att kontrollera att artefakten fungerade som planerat och om upplösningarna var passande för algoritmerna. Från pilottestet genomfördes sedan diverse korrigeringar, bland annat intervallet av upplösningar. För datainsamlingen användes två olika datorer, varav valet av datorer utgick från en begränsad uppsättning av tillgängliga enheter. Utifrån vad som fanns tillgängligt valdes därmed datorer som inte bör vara alltför identiska men som också har hårdvaror som är relevanta i dagsläget. Därmed valdes en stationär dator och en bärbar dator med följande hårdvara:

	Stationär	Bärbar
CPU	Intel i9-13900K, 3000 Mhz, 24 kärnor, 32 trådar	Intel i7 13700H 2400MHz 14 kärnor, 20 trådar
GPU	Nvidia Geforce RTX 4070 Ti (12GB VRAM)	Nvidia RTX 4060 Laptop GPU 8GB VRAM
Minne	32 GB RAM 4800 MT/s	16 GB Ram 5200 MT/s
Operativsystem	Windows 11	Windows 11

Tabell 2: Hårdvaran som brukades för experimentet.

Inför datainsamlingen förbereddes samtliga datorer för att minska påverkan från störande variabler genom att stänga ner flera applikationer som agerade i bakgrunden. När experimentet genomfördes testades implementationen för *Gabor Noise* separat från resterande, eftersom testmiljön inte kunde hantera varierande problemstorlekar. Som beskrivet tidigare överförde klassen för datahantering rådatan till respektive fil. CSV-filerna laddas sedan upp i ett Google Kalkylark för bearbetning och visualisering med inbyggda verktyg.

4.4 Resultat

Resultaten från experimentet presenteras i flera olika grafer. Det presenteras en form av grafer bestående av linjediagram för minimum, median och maximum. Vidare presenteras även grafer som visar medianvärden från flera brusimplementationer för att underlätta jämförelser. För samtliga grafer som inkluderar data för *Gabor Noise* uppnår upplösningen 256×256 medan grafer med endast implementationer för *Perlin Noise* och *Simplex Noise* uppnår 768×768 .

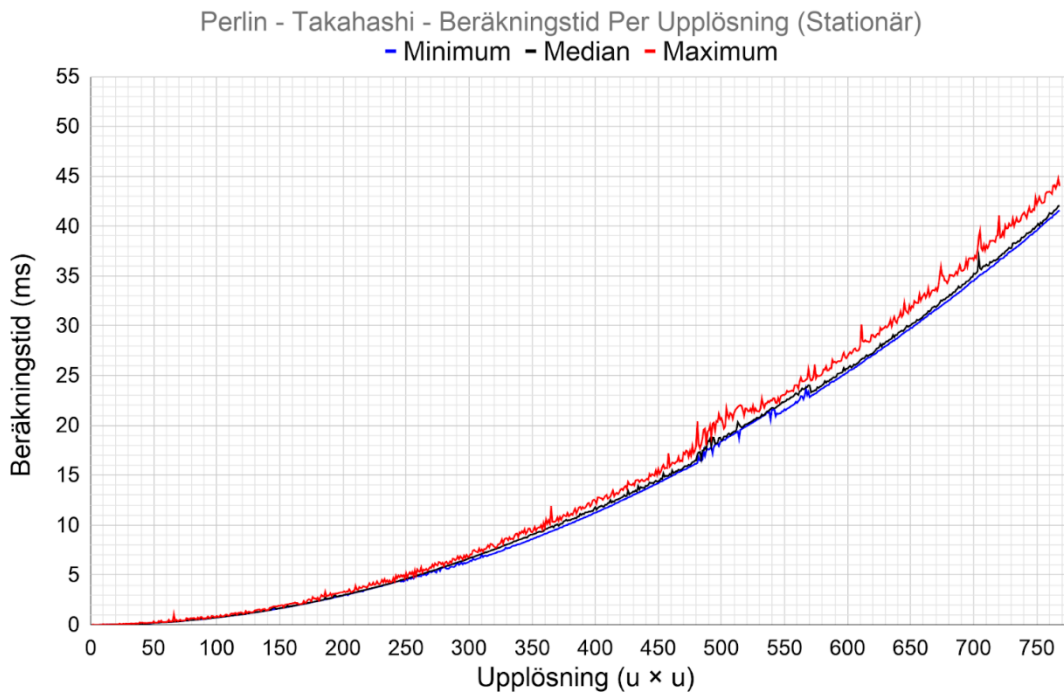
4.4.1 Exekveringstid per upplösning

Exekveringstid per upplösning syftar till att redovisa en fördjupad bild av hur brusimplementationernas prestanda utvecklas med upplösningen. Graferna fördelas

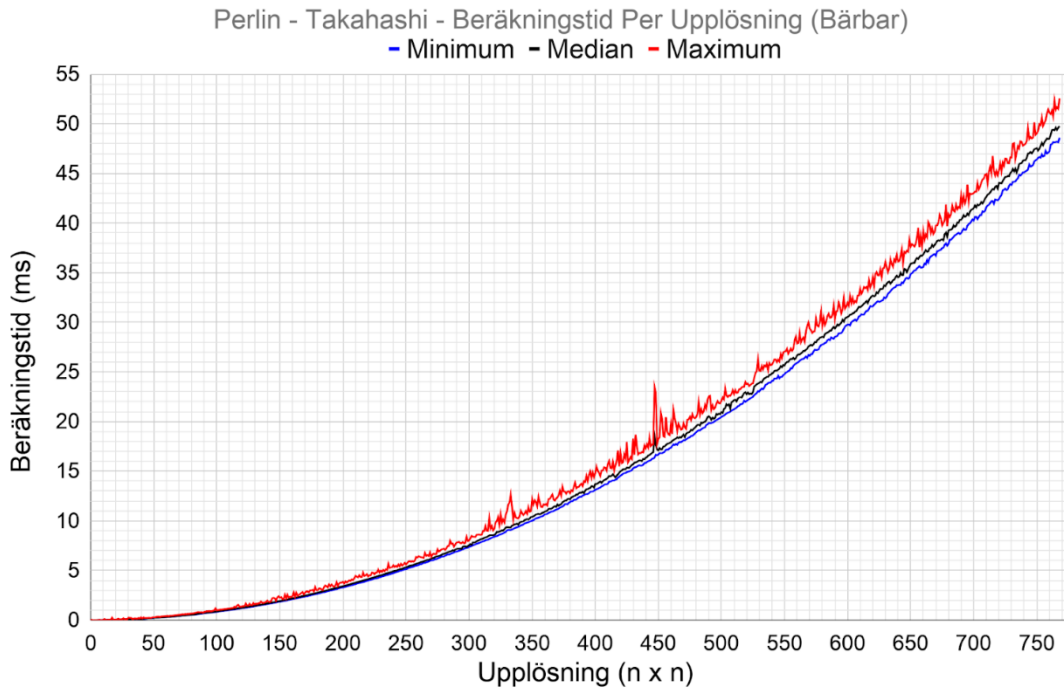
utifrån hårdvaran och inkluderar värden för minimum, median och maximum. Det främsta syftet med att inkludera alla värden är för att kunna analysera eventuell spridning i datan utan att dölja relevant data.

4.4.1.1 Perlin Noise

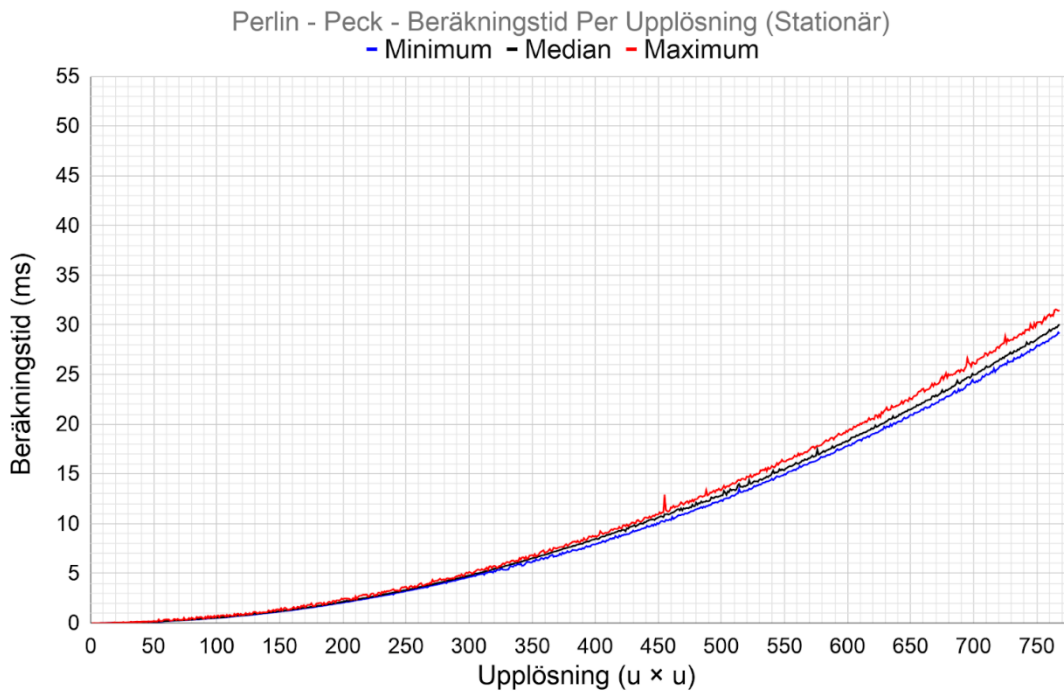
Figureerna 10 och 11 visualiserar datan för *Perlin*-implementationen av Takahashi (2015) medan figur 12 och 13 visualiserar *Perlin Noise* av Peck (2016). Figur 10 och 12 presenterar data från den stationära datorn medan figur 11 och 13 är från den bärbara datorn.



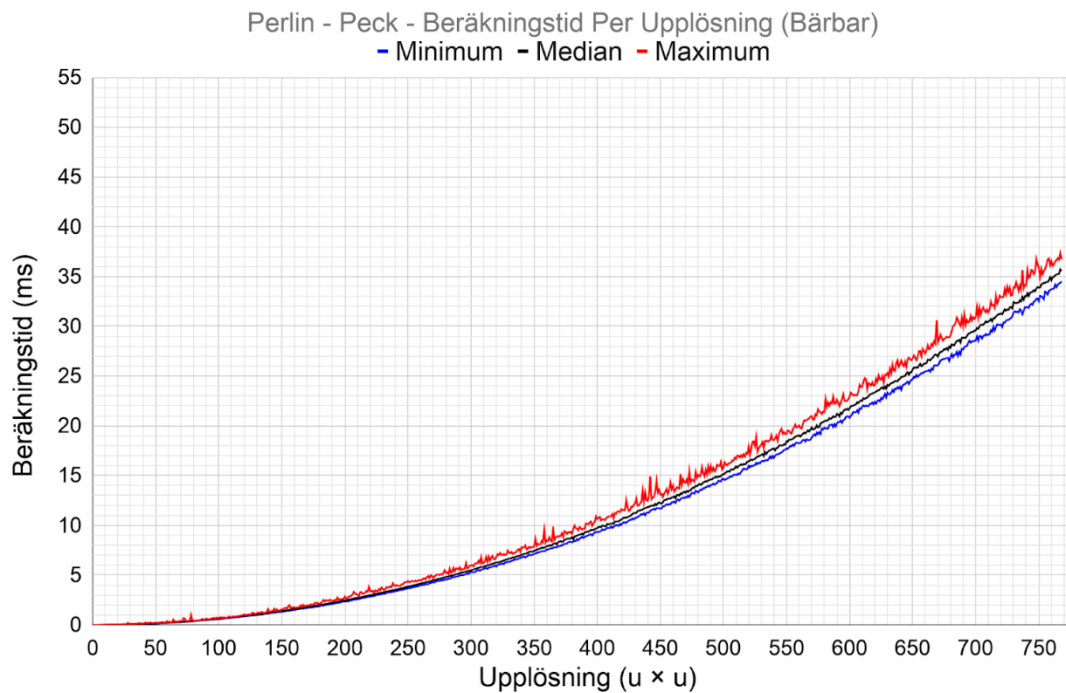
Figur 10: Grafen visualiserar resultaten för *Perlin*-implementationen av Takahashi (2015) från en stationär dator.



Figur 11: Grafen visualiserar resultaten för Perlin-implementationen av Takahashi (2015) från en bärbar dator.



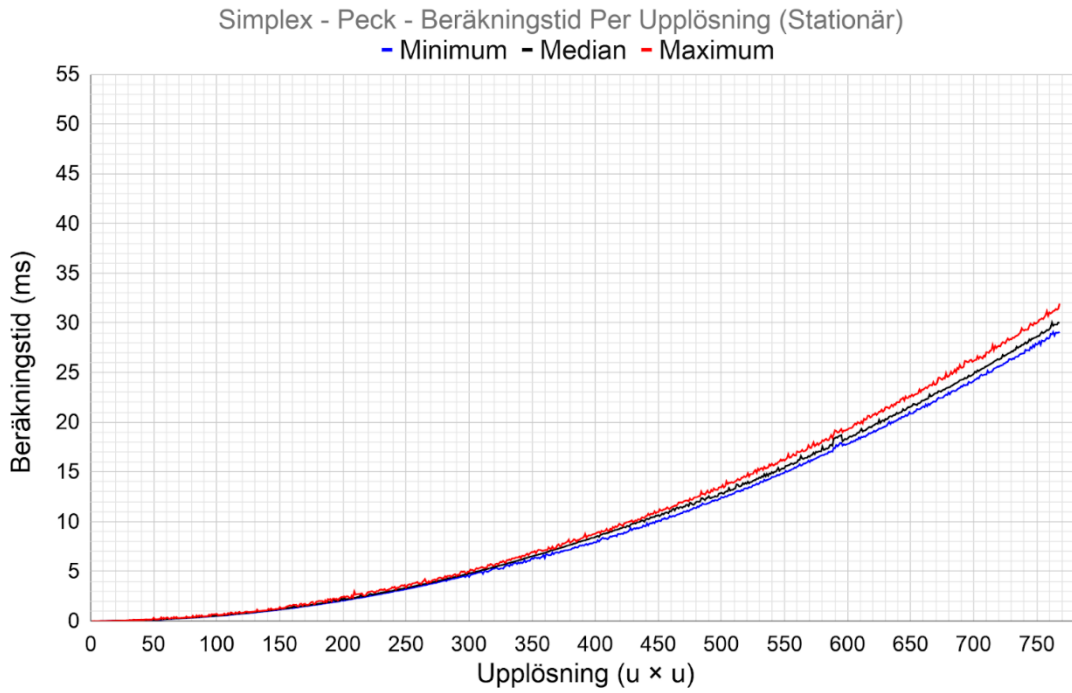
Figur 12: Grafen visualiserar resultaten för Perlin-implementationen av Peck (2016) från en stationär dator.



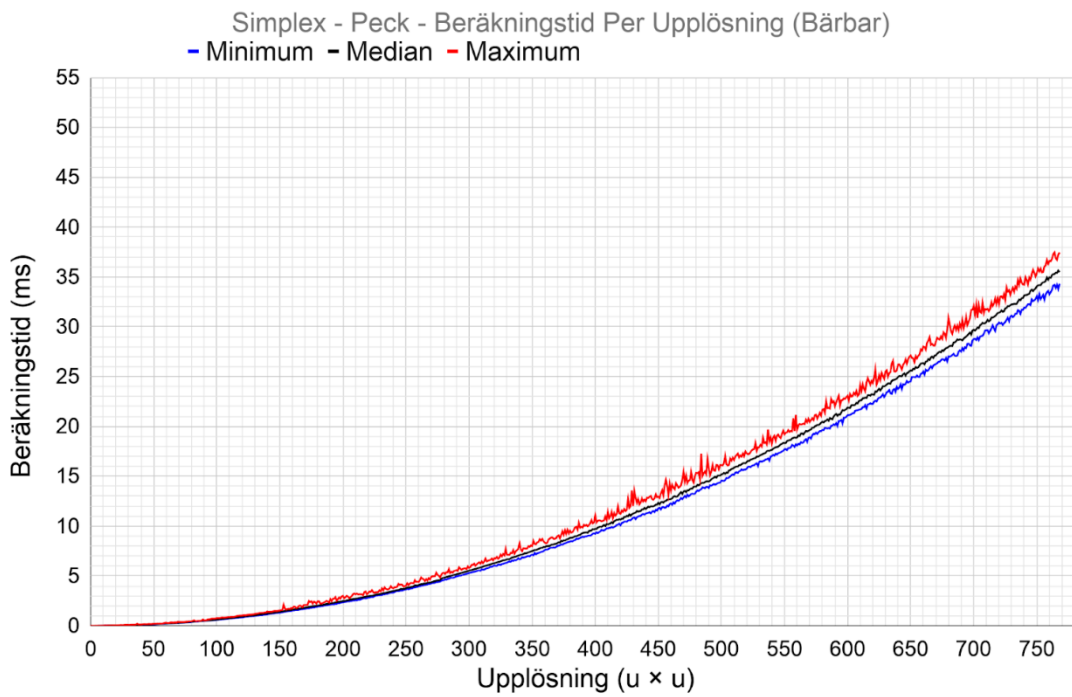
Figur 13: Grafen visualiserar resultaten för Perlin-implementationen av Peck (2016) från en bärbar dator.

4.4.1.2 Simplex Noise

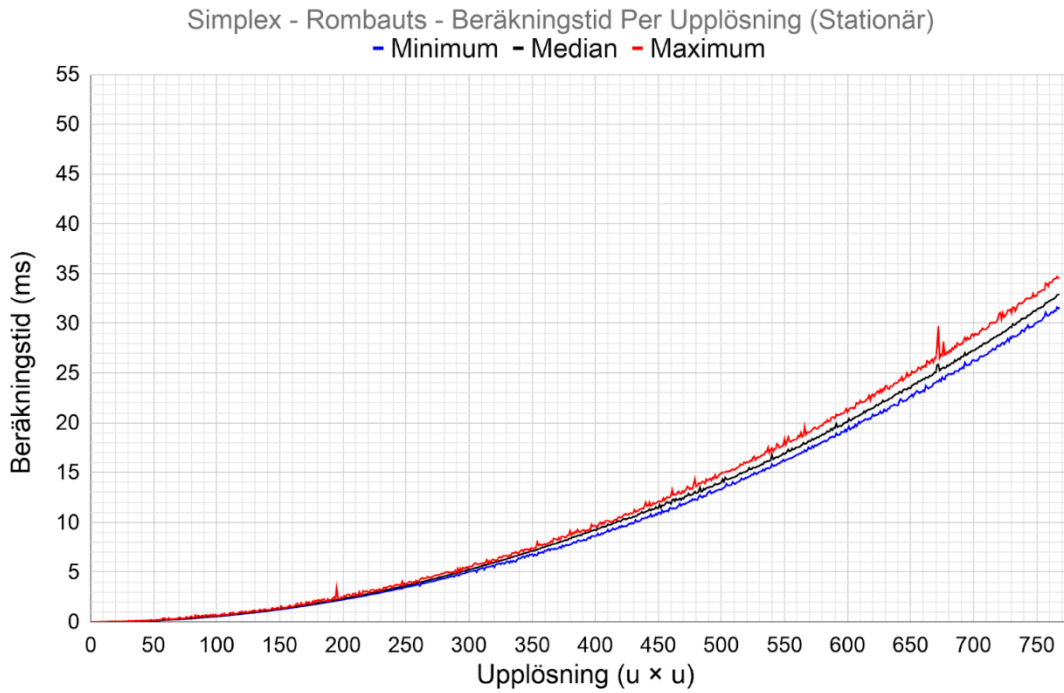
Figur 14, 15, 16 och 17 presenterar resultaten från alla implementationer av *Simplex Noise* och redogör för hur exekveringstiden utvecklas med upplösningen. Graferna i figur 14 och 15 visualiserar implementationen av Peck (2016) medan figur 16 och 17 presenterar implementationen av Rombauts (2014). Datan från figur 14 och 16 är insamlad från den stationära datorn varav figur 15 och 17 är från den bärbara datorn.



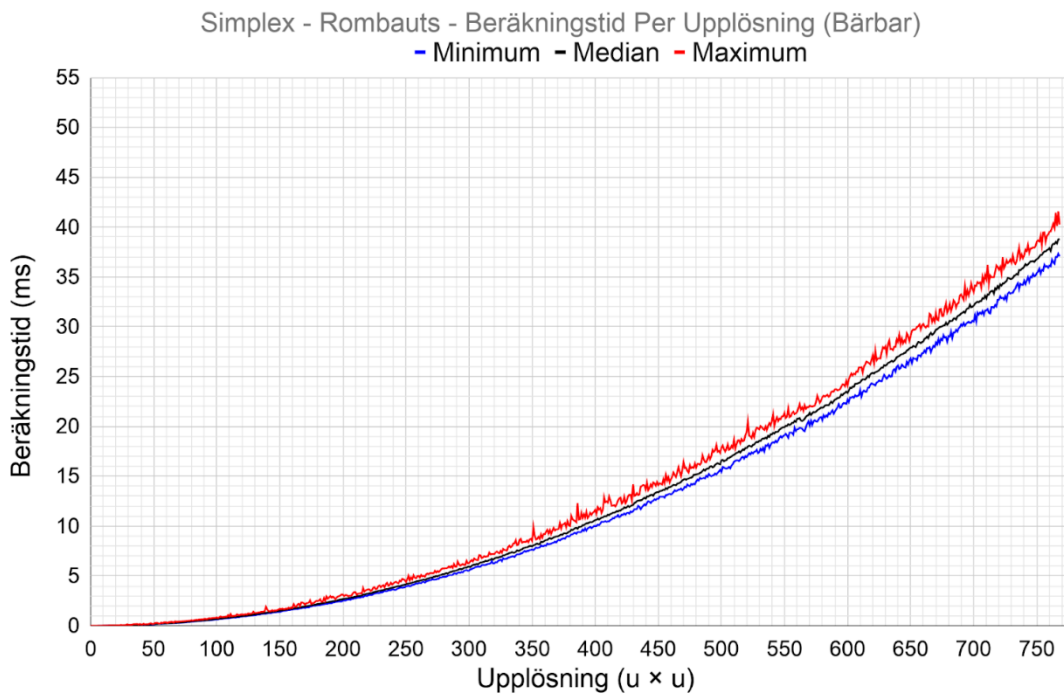
Figur 14: Grafen visualiserar resultaten för Simplex-implementationen av Peck (2016) från en stationär dator.



Figur 15: Grafen visualiserar resultaten för Simplex-implementationen av Peck (2016) från en bärbar dator.



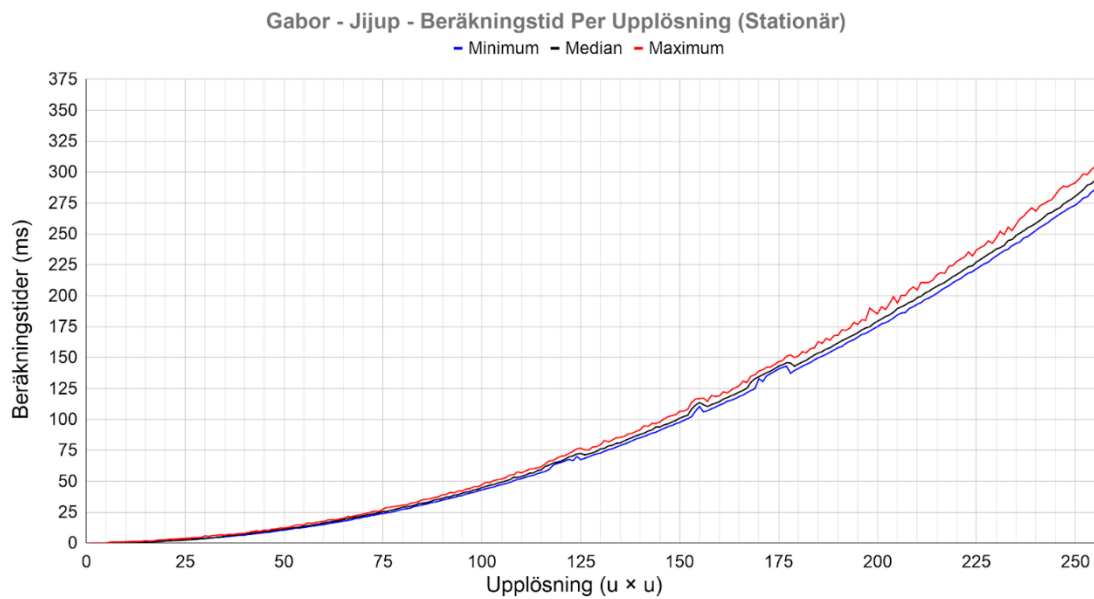
Figur 16: Grafen visualiserar resultaten för Simplex-implementationen av Rombauts (2014) från en stationär dator.



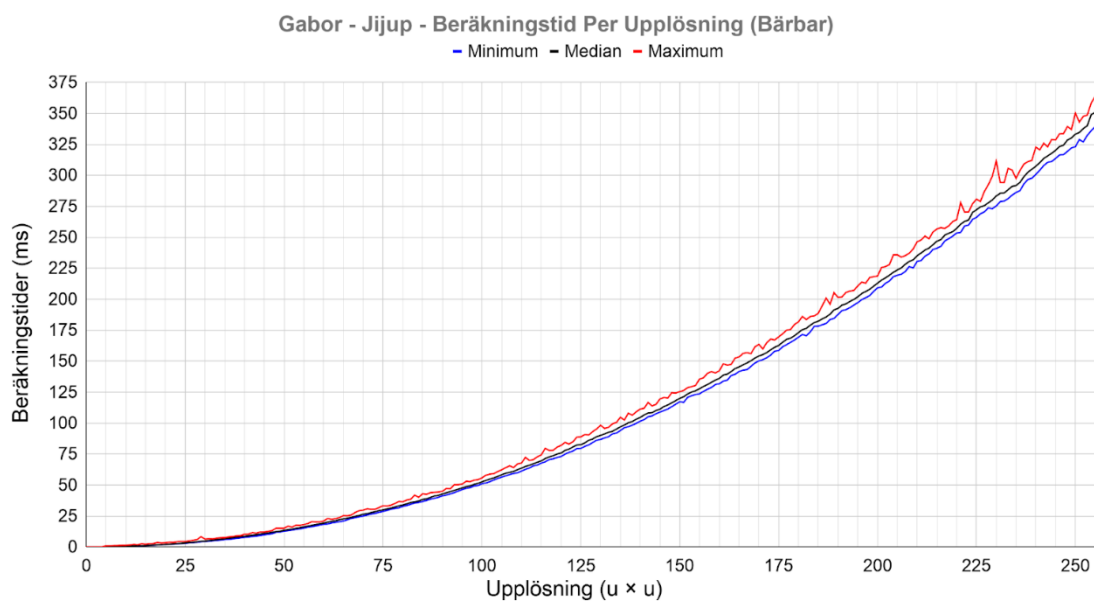
Figur 17: Grafen visualiserar resultaten för Simplex-implementationen av Rombauts (2014) från en bärbar dator.

4.4.1.3 Gabor Noise

För *Gabor*-implementationen av Jijup (2020) presenteras exekveringstid per upplösning i figur 18 och 19 där datan för figur 18 är från den stationära datorn och 19 är från den bärbara datorn.



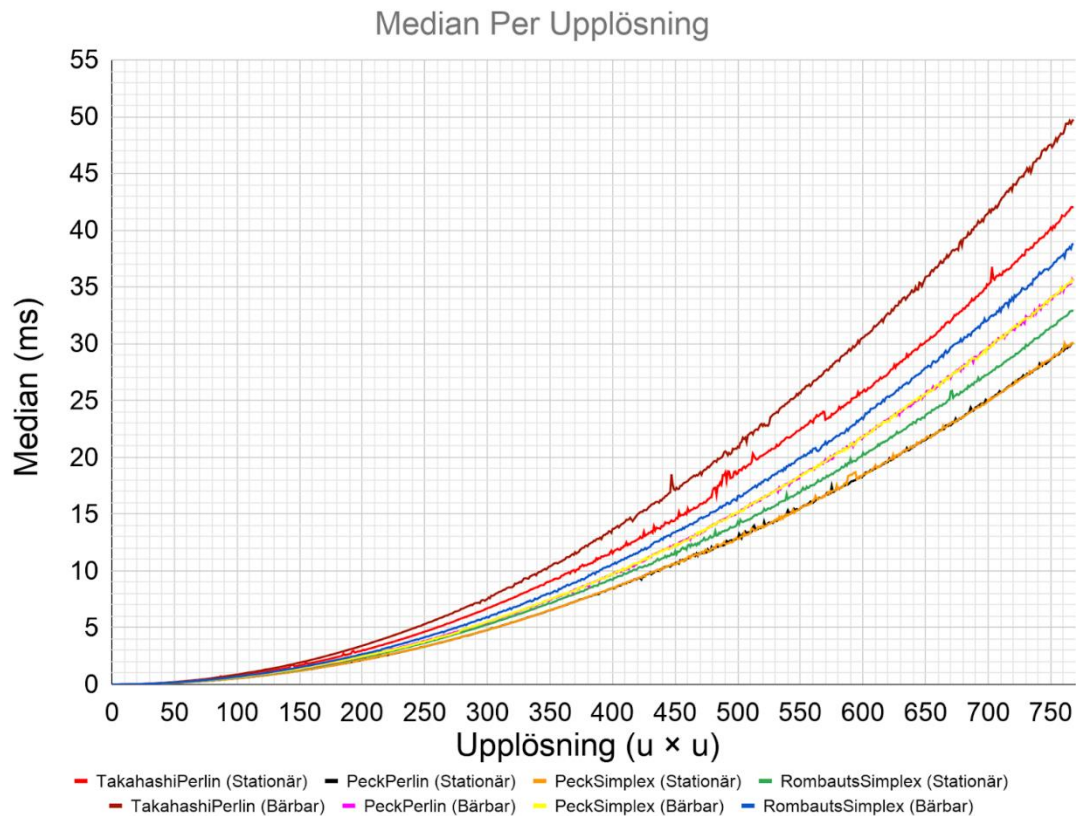
Figur 18: Grafen visualiserar resultaten för Gabor-implementationen av Jijup (2020) från en stationär dator.



Figur 19: Grafen visualiserar resultaten för Gabor-implementationen av Jijup (2020) från en bärbar dator.

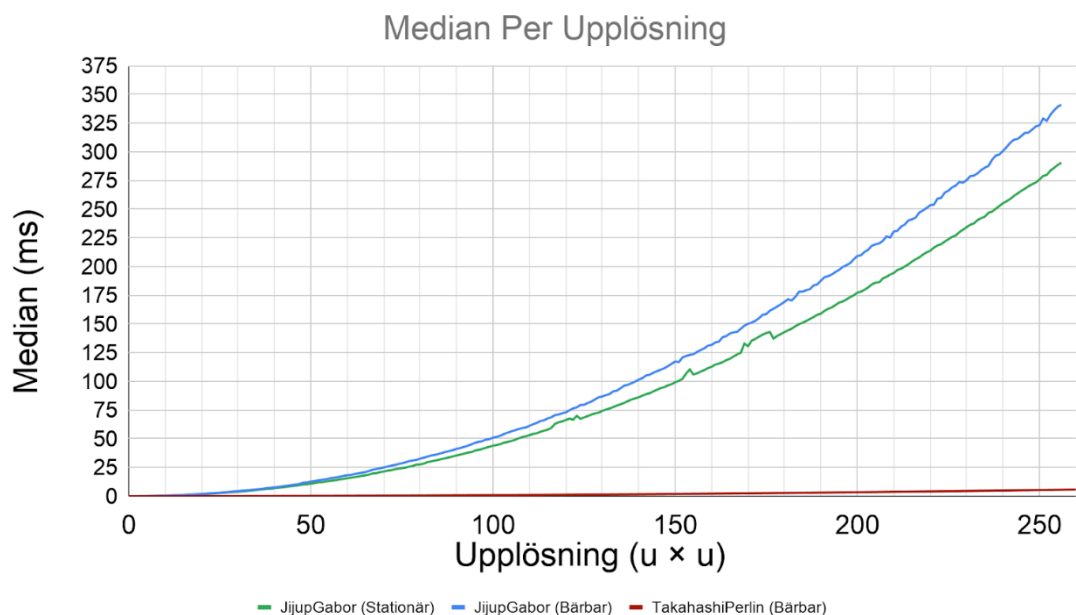
4.4.2 Median

I figur 20 och 21 presenteras grafer över hur medianen ökar med upplösningen. Grafen inkluderar resultatet från flera implementationer från både stationär och bärbar dator. Medianen från alla implementationer sammanställs i en graf som bidrar till jämförelse mellan implementationerna. Figur 20 innehåller medianen från samtliga *Perlin Noise* och *Simplex Noise* implementationer och tydliggör hur exekveringstiden utvecklas beroende på upplösning i relation till andra testade implementationer. Implementationen av *Perlin Noise* och *Simplex Noise* av Peck (2016) på den stationära datorn har bäst resultat, medan *Perlin*-implementationen av Takahashi (2015) från den bärbara datorn har högst median längs grafen.



Figur 20: Visar medianen per upplösning. Grafen inkluderar bärbara och stationära implementationer för Perlin Noise och Simplex Noise.

Figur 21 fokuserar på hur *Gabor Noise* utvecklas på bärbar och stationär dator i relation till varandra, men också jämfört med det sämsta resultatet från resterande implementationer för att se hur mycket sämre *Gabor Noise* presterar. Det syns en tydlig skillnad mellan medianen för implementationerna för *Gabor Noise* av Jijup (2020) och resterande. Överlag presterar *Gabor*-implementationen på den bärbara datorn med högst median och därmed sämst.



Figur 21: Visar medianen per upplösning. Grafen inkluderar data från den bärbara och stationära datorn för implementationen av *Gabor Noise*, samt data från den bärbara datorn för *Perlin*-implementationen av Takahashi (2015).

4.5 Analys

Ett analysarbete genomfördes för att tolka, förklara och utvärdera resultatet för att bidra till en fördjupad förståelse kring prestandan. I analysen jämförs implementationerna och algoritmerna centralt utifrån dess lämplighet för realtidsanvändning, men också gentemot varandra och tidigare arbeten. Vidare undersöks diverse mönster och beteenden som uppkommer i resultatet för att klargöra bakomliggande anledningar till dessa. Slutligen ämnar fynden inom analysarbetet att jämföras med förväntningarna beskrivna i hypotesen och för att besvara frågeställningen.

Merparten av resultatet består av graferna för exekveringstid per upplösning (figur 10 - 19), varav samtliga uppvisar liknande beteenden för hur värdena utvecklas. Det bör dock noteras att värdena inom grafen inte har samma utveckling utan skiljer sig gentemot varandra. Kurvan för maximum är mer volatil med generellt ett längre avstånd till medianen jämfört med kurvan för minimum som generellt är närmare medianen. Utvecklingen för maximivärdet blir främst anmärkningsvärt vid högre upplösningar vilket kan komma av att det finns ett större tidsspänn, som ökar möjligheten att inkludera störande variabler. Dessa störningar kan vara från diverse program som fortfarande är aktiva på datorerna.

I sammanhanget av spelupplevelse så har fluktuationer betydande påverkan på spelaren enligt Liu et al. (2023) och skillnaden mellan minimum och maximum markerar den eventuella fluktuation som implementationerna producerar. Utifrån deras resultat så uppfattar inte spelaren någon negativ påverkan om spridningen är inom 0 till 4 millisekunder. Alla implementationer bortsett från *Gabor Noise* är inom detta spann vid den högsta upplösningen. Trots den lägre maximala upplösningen för *Gabor Noise* så är skillnaden mellan lägsta och högsta noterade exekveringstiderna 24 millisekunder på den bärbara datorn. Om medianvärdet för *Gabor Noise* är i närheten av det som implementationen av Takahashi (2015) uppnådde vid högsta upplösningen i figur 11, så skiljer det ungefär 5 millisekunder.

Trots förekomsten av flera implementationer av samma brusalgoritm utvecklas dessa olika, vilket syns i figur 20 och figur 21. Varken Takahashi (2015) eller Rombauts (2014) implementeringar presterar med lika korta exekveringstider som båda implementationerna av Peck (2016). Faktorer som kan påverka är bland annat vilken variant och version av algoritmen som implementerats. Ett exempel på detta är förbättringar som Lagae, Lefebvre och Dutré (2011) och Perlin (2002) genomförde på de ursprungliga algoritmerna (Lagae et al. 2009; Perlin 1985). Exempelvis kan både Perlin (1985) och Perlin (2002) trots skillnader användas som möjlig grund för att utveckla implementationer av *Perlin Noise*. Detta behöver dock inte vara enda förklaringen till att resultatet för båda implementationerna av Peck (2016) är nästintill identiska även fast implementationerna tillhör olika brusalgoritmer. En ytterligare potentiell anledning är hur implementeringen genomförs via programmering, varav diverse optimeringstekniker eller designval kan påverka.

Eftersom arbetet inkluderar flera implementationer av samma algoritmer så bidrar detta till en annan bredd i resultaten eftersom ett spann av exekveringstider uppkommer per

algoritm. Om man jämför detta med *Gabor Noise* som endast har en implementering så är det märkbart i resultaten att den enskilda implementeringen kan ha en mer betydande roll vid valet av algoritm, för specifikt prestanda.

Resultatet för *Gabor Noise* har den snabbast växande kurvan (figur 21) och särskiljer sig gentemot samtliga implementationer då skillnaden snabbt blir stor. Detta är delvis rimligt utifrån hur algoritmen fördelar kärnor över flera celler men förändring av nuvarande parametrar kan eventuellt påverka hur utvecklingen sker. Lagae et al. (2009) beskriver ett tydligt samband mellan prestanda och kvalitet som hanteras av parametrar anknutet till densiteten av impulser i *Gabor Noise*.

Förhållandet mellan kurvorna från den stationära och den bärbara datorn är mycket lika, något som skulle kunna bero på att de två datorerna har för lika specifikationer. Det är dock tydligt utifrån resultaten att den stationära datorn presterar med kortare exekveringstider. Övergripande har den bärbara datorn 17 % högre exekveringstider än den stationära om resulterande medianer från implementationerna jämförs.

I många av graferna existerar några visuella artefakter som snedvrider och försvårar tolkningen av brusimplementationerna. Samtliga resultatgrafer visar att alla implementationer stiger exponentiellt, som sannolikt beror på att problemstorlekarna har en kvadratisk ökning. Om problemstorlekarna däremot hade ökat linjärt skulle kurvorna eventuellt blivit mer linjära. Vidare kan resultaten för *Gabor Noise* (figur 18, 19 & 21) uppfattas som mjukare jämfört med resterande implementationers resultat, vilket troligtvis beror på att mindre data behöver presenteras i graferna. Skillnaden i data beror på att den högsta problemstorleken för *Gabor Noise* utgår ifrån en lägre maximal upplösning på 256×256 än resterande implementationer som uppnår 768×768 i upplösning.

Från resultatet kan flera avvikande fluktuationer och höjningar skådas, bland annat i figur 10 mellan upplösningarna 470×470 till 550×550 och figur 21 vid upplösningen 175×175 . Dessa ökningarna av exekveringstid kan bero på bakgrundsaktivitet eftersom liknande beteende inte återkommer i resultaten med den bärbara datorn.

Överlag överensstämmer hypotesen med resultaten som har erhållits av experimentet. Sammantaget presterar implementationerna av *Simplex Noise* något bättre än *Perlin Noise* trots den låga dimensionen som används i experimentet. Resultatet visar att implementationen av *Gabor Noise* har högst exekveringstid av alla inkluderade implementationer.

Det är komplicerat att besvara frågeställningen i både konkreta och generella slutsatser för *Perlin* och *Simplex Noise* utifrån resultatet eftersom variationen är stor beroende på brusimplementation. Trots stora skillnader mellan implementationerna presenterar resultatet att *Perlin* och *Simplex Noise* kan rimligen användas inom realtidsbaserade lösningar vid lägre upplösningar. Eftersom *Gabor Noise* resultat visar på en betydligt snabbare utveckling än resterande algoritmer, så är också utrymmet mer begränsat för när den kan användas i realtidsbaserade lösningar.

5 Sammanfattning och diskussion

I detta kapitel sammanfattas studien i sin helhet och det genomförs en diskussion kring studien ur ett bredare perspektiv. Diskussionen leder till en tydligare förståelse för vilka sammanhang resultaten gäller samt vilka identifierade brister. Inom kapitlet uppmärksammas också samhälleliga och etiska perspektiv i anknytning till studien. Avslutningsvis beskrivs framtida arbete med bearbetning av brister i arbetet men också passande för utveckling av studiens innehåll.

5.1 Sammanfattning

Brusalgoritmer är vanligt förekommande inom diverse steg vid skapandet av spel och förekommer ofta inom procedurrell terränggenerering vilket gör dem relevant för realtidsbaserade implementationer. Inom procedurrell generering kan brusalgoritmer bidra till dynamiska och levande spelvärldar, samt relativt låga exekveringstider jämfört med fysikbaserade implementationer. Med den generellt låga exekveringstiden används brusalgoritmer frekvent inom realtidsbaserade GPU- och CPU-implementationer.

Inom spel finns ett behov av att bibehålla en relativt låg exekveringstid. Brusalgoritmer är utvecklade utifrån olika ändamål och sammanhang vilket leder till variationer i tillvägagångssättet för att beräkna och konstruera respektive brus. Därtill kan det finnas ett värde i vetenskapen om vilka sammanhang som olika algoritmer är lämpliga för.

Detta arbete utförde ett experiment med olika upplösningar som problemstorlekar för att samla in exekveringstider från olika brusimplementationer. Vidare var exekveringen begränsad till att endast ske på CPU:n. De brusalgoritmer som testades är *Perlin Noise*, *Simplex Noise* och *Gabor Noise*. Resultatet jämfördes med hur varje implementations exekveringstid förhöll sig till övriga implementationer, och att algoritmerna analyserades utifrån deras lämplighet i realtidsbaserade miljöer.

För att stärka trovärdigheten i resultaten från experimentet genomfördes anpassningar. Flera iterationer per problemstorlek användes för samma implementation och datan samlades även in på två datorer med olika hårdvaror. Antalet problemstorlekar är många och utgår ifrån hur höjdkartor vanligen används med brus. Nödvändiga begränsningar har introducerats inom flera delar av arbetet för att göra omfånget mer hanterbart. Bland annat innebar det uteslutandet av flera oktaver och antalet brusimplementationer begränsades till totalt fem.

Från resultatet presenterades det som komplicerat att kunna dra generella slutsatser om algoritmernas prestanda och att den enskilda implementationen hade avgörande påverkan på prestandan. Implementationerna för *Perlin* och *Simplex Noise* presterade i sammanhanget av *Gabor Noise* betydligt bättre och inom ramen för realtid. Vidare blev resultatet för *Gabor Noise* långsamt gentemot resterande och möjligheten att använda den i realtid är avsevärt begränsad.

5.2 Diskussion

För arbetet genomförs en diskussion som ämnar att beskriva studien utifrån dess helhet. Eventuella brister i studien diskuteras, vilka konsekvenser bristerna har och vilka förbättringar som kan genomföras. Bristerna påverkar trovärdigheten i resultatet men

tidigare studier inkluderas för att avgöra resultatens rimlighet. Vidare diskuteras resultatets generaliserbarhet och tillämpningar i ett större sammanhang.

Som tidigare har redogjorts har omfånget för studien begränsats i många avseenden, vilket innebär brister i metodiken och genomförandet som påverkar resultatets trovärdighet och tillämpning. Huvudsakligen utgör implementeringarna som inkluderats i studien endast en liten del av alla tillgängliga implementationer för diverse brusalgoritmer. Eftersom resultaten för implementationerna skiljer sig gentemot andra av samma algoritm, blir det svårt att genomföra en konkret jämförelse av brusalgoritmerna.

Vad gäller algoritmen *Gabor Noise* finns flera bristande aspekter. För det första representeras algoritmen endast av en implementering, varav problematiken uppkommer från ett bristande urval av tillgängliga implementationer med *Gabor Noise*. Därmed kan ingen redogörelse göras för hur stora eventuella prestandaskillnader är mellan diverse implementationer. Detta innebär att underlaget är mindre för denna algoritm än de övriga i studien. För det andra hanterar *Gabor Noise* fler parametrar, som enligt Lagae et al. (2009) har en tydlig korrelation mellan prestanda och den visuella kvaliteten vid generering av bruset. Att redovisa resultatet för endast en definierad uppsättning parametrar beskriver inte hela variationens djup som potentiellt existerar med *Gabor Noise*.

Det är tydligt i resultatet att det förekommer bakomliggande variabler som leder till märkbara höjningar i exekveringstiden. Som tidigare beskrivits kan en anledning vara bakgrundsaktivitet via externa program på datorn, detta trots att datorerna förbereddes genom att avsluta diverse program i bakgrunden. Framst påverkar detta den lokala tillförlitligheten i resultatet, medan övergripande resultat fortfarande visar tydliga trender och jämförelser. En åtgärd för att minska påverkan från störande variabler med datorerna skulle kunna vara att dedikera en isolerad CPU-kärna, som experimentet kan använda för att minska störningar från andra applikationer (de Oliveira, Casini & Cucinotta 2024).

Hela experimentet genomfördes med implementationer ämnade för 2D-höjdkartor och utforskade inga ytterligare dimensioner. Valet av 2D är primärt motiverat utifrån att minska arbetets omfång. För algoritmer som beskrivs som mer beräkningseffektiva på högre dimensioner innebär detta att de felaktigt representeras utifrån avsaknad av jämförelse mellan dimensioner.

I analysen presenterades koncepten kallstart och uppvärmt tillstånd varav exekveringstiden från enbart kallstart eller uppvärmt tillstånd är användbart men bör inte blandas (Akinshin 2019, s.49-51). I studiens fall kombineras båda vilket innebär initialt höga kallstartsvärden som övergår till värden från uppvärmda körningar. Utifrån hur det aktuella arbetets artefakt är strukturerad skulle en förberedande process (eng. *warm-up*) i testmiljön kunna introduceras för att endast erhålla exekveringstider av uppvärmda algoritmer. Exekveringstider utifrån kallstart är främst relevant enligt Akinshin (2019, s.49-51) när algoritmer körs fåtal tillfällen, vilket överlag inte överensstämmer med hur brusalgoritmer vanligen brukas i realtidssammanhang.

Från resultatet kan vissa prestandaskillnader och fluktuationer mellan implementationerna observeras när dessa exekveras utifrån olika hårdvaror, men inget oväntat samband till hårdvaran. Trots ändamålet i att nyttja två distinkt olika datorer kan hårdvaran vara för likartad, bland annat är processorerna från samma generation

även fast de inte är identiska. Förbättringar här innebär att inkludera hårdvara som är mer distinkt från moderna Intel CPU:er men med liknande relevans för att brukas i praktiken.

Konverteringsarbetet av implementationerna bör beskrivas som en brist, trots att det möjliggör att implementeringarna kan prövas i testmiljön. Att manuellt översätta från ett programmeringsspråk till ett annat introducerar potentiella variationer i implementationen. Detta förstärks troligen även av en begränsad kunskap kring programmeringsspråket som översätts. Dessa variationer är potentiellt inte ekvivalenta i sammanhanget av prestanda och därmed en möjlig faktor till minskad reproducerbarhet av resultatet. För att praktiskt kunna genomföra arbetet är detta en nödvändighet, men det kan vara en utmaning att avgöra när den konverterade implementationen övergår till att inte längre vara tillräckligt lik den ursprungliga koden.

Från arbetet av Lagae, Lewis och Dutré (2011) presenteras en jämförelse av prestanda mellan *Perlin Noise*, *Wavelet Noise* och *Gabor Noise* utifrån GPU:n och skillnaden är stor mellan algoritmerna. En intressant detalj från denna studie är hur *Gabor Noise* presterade olika beroende på parametrar, med exekveringstider på 4 ms eller 7 ms medan *Perlin Noise* presterade runt 0,3 ms på samma upplösning. Studien av Vitacion och Liu (2019) framför relevanta jämförelser för brusalgoritmerna *Perlin Noise* och *Simple Noise* fast med flera oktaver, varav *Perlin Noise* likt resultatet från denna studie presterar lite sämre gentemot *Simple Noise*. Slutligen redogör Strand och Prestberg (2025) liknande förhållanden mellan algoritmerna som Vitacion och Liu (2019) men uttrycker att implementationen har stor betydelse för prestandan. Resultatet från dessa arbeten överensstämmer när det gäller hur algoritmerna presterar i relation till varandra samt implementationernas individuella påverkan på resultatet.

Från tidigare arbeten sker experimenten utifrån andra förhållanden än inom denna studie som sedermera kan påverka validiteten bakom jämförelsen av resultaten. Främst beror detta på att brusalgoritmerna prövats i sammanhanget av GPU:n (Lagae, Lewis & Dutré 2011) eller med flera oktaver (Vitacion & Liu 2019; Strand & Prestberg 2025). Sedan har det beskrivits tidigare att studien exkluderar genereringen av en höjdkarta och beräknar endast respektive pixel jämfört med Vitacion och Liu (2019) som också analyserade minnets påverkan på brusimplementationernas prestanda.

Gränsdragningen för när en algoritm och dess tillhörande implementationer lämpas för realtidsapplikering utgår ifrån hur exekveringstiderna ökar per upplösning. Denna aspekt påverkas av hårdvaran, samt storleken på applikationens budget för exekveringstiden. Därmed är sammanhanget kring när och hur implementeringar används viktigt. Främst redogör resultatet en anvisning på dugliga spann av upplösningar. Vidare beskrivs det i arbeten som exempelvis Tan et al. (2022) riktlinjer för diverse spelgenrer och när spelaren börjar påverkas negativt av låga FPS. Som tidigare nämnt så påverkar också fluktuationer mellan *frames* (Liu et al. 2023), men utifrån denna studies resultat är det potentiellt märkbart först vid högre exekveringstider (figur 10-19).

Resultatets praktiska anknytning är något bristfällig. Exempelvis i spelet Minecraft (2009) skapas terräng utifrån lågupplösta delområden, och i detta spel laddas flera delområden in kontinuerligt utifrån vad som anses vara inom en definierad radie. Detta praktiska exempel visar att låga upplösningar av höjdkartor med brus används, men att det genereras i flera uppsättningar. Så som experimentet är strukturerat nu beräknas

endast en höjdkarta per tidtagning vilket inte reflekterar den beskrivna praktiska användningen. Ytterligare tillämpas *Fractal Noise* oftast inom spel för ökad detaljnivå, vilket genomförs med olika oktaver av brus som sedan sammanställs (Lagae et al. 2010). Detta är ännu ett exempel på hur den praktiska implementeringen av brus kan använda flera höjdkartor, medan experimentet endast producerar och redogör resultat för alla pixlar i en enskild höjdkarta.

Flera optimeringsprocesser kan utnyttjas i samband med brusalgoritmer för att reducera exekveringstiden. GPU:n används ofta för att beräkna flertalet pixlar samtidigt eftersom varje pixel kan beräknas separat (Lagae et al. 2010). Parallellisering kan också genomföras på CPU:n med samma ändamål. Ett annat exempel är LOD (eng. *Level of Detail*) som används för att minska detaljen på objekt beroende på avståndet till spelaren. Detta innebär att färre beräkningar behöver utföras då det inte längre finns lika mycket detaljer. Dessa optimeringsprocesser kan kombineras i praktiken för att reducera antalet beräkningar, men också den totala exekveringstiden. Det innebär alltså att exekveringstiderna som erhållits från detta experiment sannolikt kan förbättras med optimeringstekniker.

5.3 Samhälleliga och etiska aspekter

För studien har ett experiment utförts utan deltagare och berör endast testning av algoritmer på datorer. Datan som behandlas är endast exekveringstid och därmed varken hanteras eller sparas personlig data tillhörande någon individ. Studien ämnar heller inte att på något vis med externa implementationer sprida eller orsaka skada vid brukandet av inkluderade algoritmer. Slutligen har inga resultat varken förfalskats eller ämnats till att orsaka lidande gentemot en annan individ.

Resultaten är inte begränsade till att enbart vara relevant inom spelutveckling. Brusets breda applicerbarhet innebär att områden och industrier utanför spelutveckling kan tillhandahålla diverse resultat från studien. Trots att exekveringstid har stor betydelse för realtidsbaserade lösningar kan det vara relevant också i sammanhanget energiförbrukning. Genom att minska mängden arbete som behöver utföras reduceras mängden konsumerad energi (Corral-García, Lemus-Prieto, González-Sánchez & Pérez-Toledano 2019), som konsekvent kan bidra till minskade utsläpp. Vidare kan val av bättre lämpade algoritmer leda till att prestandakraven på hårdvaran minskar. Med lägre systemkrav på spel, applikationer och produkter som utvecklas innebär potentiellt en ökning av tillgänglighet för de som tidigare inte hade möjligheten. Ett ytterligare tillämpningsområde är inom användarinteraktion som inte enbart existerar inom spelindustrin. Brus kan bidra med lägre exekveringstider än exempelvis fysikbaserade lösningar och agera som komplement för att bidra till en förbättrad användarupplevelse.

En av de främsta fördelarna med brusgenerering är den automatiserade processen som minskar den manuella arbetsbördan men detta kan också innebära mindre arbete för de individer som livnär sig på denna typ av arbete. Detta kan ge upphov till konflikter mellan den enskilda individens bästa och utvecklingen av produktens bästa.

5.4 Framtida arbete

Under arbetets gång har eventuella brister framträtt som vid ytterligare arbete borde behandlas. Dessa förbättringar inkluderar uppvärmning av implementationerna innan

mätningen av exekveringstiden, ytterligare en *Gabor*-implementation, en dedikerad kärna på CPU:n för experimentet och slutligen större variation i hårdvaran.

Studien kan även utvidgas för att bidra till en bredare bild kring prestanda för brusalgoritmer. I detta arbete inkluderas tre olika brusalgoritmer men variationen är stor och ytterligare jämförelser kan utföras med andra algoritmer och implementationer. Vidare påpekar litteraturen att algoritmerna i många fall presterar annorlunda i andra dimensioner än 2D och därmed borde en rimlig utveckling av studien inkludera högre dimensioner. Dessutom kan vissa algoritmer utforskas djupare kring diverse parametrar som i fallet av *Gabor Noise* där denna studie endast har en uppsättning parametrar.

Studien är främst anpassad för att avgöra prestandaskillnader mellan brusalgoritmer och inte i hur dessa algoritmer presterar i praktiken. En rimlig utveckling av arbetet skulle kunna inkludera påverkan från minnesallokeringar och hur detta sedan påverkar exekveringstiden för brusalgoritmer, liknande studien av Vitacion och Liu (2019). Detta syftar på att studera eventuella korrelationer mellan exekveringstid och minnesanvändning.

En intressant aspekt som hade kunnat utforskas är hur lämpligt diverse optimeringstekniker kan tillämpas för brusalgoritmer. Övergripande finns det mycket forskning som implementerar brus på GPU:n genom att parallellisera beräkningarna för alla pixlar. Liknande kan genomföras för CPU:n för att undersöka skillnaderna i prestanda och eventuella variationer i resultatet mellan sekventiell och parallelliserad exekvering CPU, samt GPU.

Referenser

Akinshin, A. (2019). *Pro. NET Benchmarking: The Art of Performance Measurement*. Apress Berkeley.

doi: 10.1007/978-1-4842-4941-3

Beiranvand, V., Hare, W. och Lucet, Y. (2017). Best practices for comparing optimization algorithms. *Optimization and Engineering*, 18(3), s. 815–848.

doi: 10.1007/s11081-017-9366-1.

Boyd, A. & Vandenberghe L (2009). *Convex Optimization*. 7 uppl, Cambridge University Press.

https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf [2026-03-05]

Büyükkşar, O., Yıldız, D. & Demirci, S. (2024). Enhancing wave function collapse algorithm for procedural map generation problem. *Nigde Omer Halisdemir University Journal of Engineering Sciences / Niğde Ömer Halisdemir Üniversitesi Mühendislik Bilimleri Dergisi*, 13(3), s. 806-814.

doi: 10.28948/ngumuh.1361413

Casini, D. (2022). A Theoretical Approach to Determine the Optimal Size of a Thread Pool for Real-Time Systems. I *2022 IEEE Real-Time Systems Symposium (RTSS)*.

Huston TX, USA 5-8 december 2022, s. 66-78.

doi: 10.1109/RTSS55097.2022.00016

Corral-García, J., Lemus-Prieto, F., González-Sánchez, J.-L. & Pérez-Toledano, M.-Á. (2019). Analysis of Energy Consumption and Optimization Techniques for Writing Energy-Efficient Code. *Electronics*, 8(10), 1192.

doi: 10.3390/electronics8101192

de Oliveira, D. B., Casini, D. & Cucinotta, T. (2024). Operating System Noise in the Linux Kernel, *IEEE Transactions on Computers*, 72(1), s. 197-207.

doi: [10.1109/TC.2022.3187351](https://doi.org/10.1109/TC.2022.3187351)

Dyrda, D., Pfaffinger, K., Belloni, C., Schacherbauer, M., Pirker, J. & Klinker G. (2022). A Time- and Space-Efficient Adaptation of the Space Foundation System for Digital Games. I *MIG '25: Proceedings of the 2025 18th ACM SIGGRAPH Conference on Motion, Interaction, and Games*. Zurich, Schweiz 3-5 december 2025, s. 1-11.

doi: 10.1145/3769047.3769056

Fischer, R., Dittmann, P., Weller, R. & Zachmann, G. (2020). AutoBiomes: procedural generation of multi-biome landscapes. *Vis Comput*, 36(7), s. 2263-2272.

doi: 10.1007/s00371-020-01920-7

Galín, E., Guérin, E., Peytavie, A., Cordonnier, G., Cani, M.-P., Benes, B. & Gain, J. (2019). A Review of Digital Terrain Modeling, *Computer Graphics Forum*, 38(2), s. 553-577.

doi: 10.1111/cgf.13657

Gustavson, S. & McEwan, I. (2022). Tiling simplex noise and flow noise in two and three dimensions, *Journal of Computer Graphics Techniques (JCGT)*, 11(1), s. 17-33,

2022

<https://jcgt.org/published/0011/01/02/> [2026-02-02]

JFokus (2022). *Reinventing Minecraft world generation by Henrik Kniberg* [video].

<https://www.youtube.com/watch?v=ob3VwY4JyzE> [2026-02-27]

Jijup (2020). OpenSN. [Git Repository] Commit: 71a301b.

<https://github.com/Jijup/OpenSN/blob/master/NoiseGeneration/Software/ProceduralNoiseUsingSparseGaborConvolution/noise.cpp> [2026-02-17]

Koulaxidis, G. & Xinogalos, S. & Gain, J. (2022). Improving Mobile Game Performance with Basic Optimization Techniques in Unity, *Modelling*, 3(2), s. 201-223.

doi: 10.3390/modelling3020014

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., Lewis, J. P., Perlin, K. & Zwicker, M. (2010). A Survey of Procedural Noise Functions. *Computer Graphics Forum*, 29(8), s. 2579–2600.

doi: 10.1111/j.1467-8659.2010.01827.x

Lagae, A., Lefebvre, S., Drettakis, G. & Dutré, P. (2009). Procedural noise using sparse Gabor convolution. I *SIGGRAPH09: Special Interest Group on Computer Graphics and Interactive Techniques Conference*. New Orleans, LA, USA 3-7 augusti 2009, s. 1-10.

doi: 10.1145/1576246.1531360

Lagae, A., Lewis, J. & Dutré, P. (2011). Improving Gabor Noise. *IEEE Transactions on Visualization and Computer Graphics*, 17(8), s. 1096–1107.

doi: 10.1109/TVCG.2010.238

Li, B. & Zhao, Z. (2012), Cloth 3D Attributes Animation Based on a Wind Noise, *Symposium on Photonics and Optoelectronics*, Shanghai, Kina 21-23 maj 2012, s. 1-3.

doi: 10.1109/SOPO.2012.6271011

Liu, S., Kuwahara, A., Scovell, J. & Claypool M. (2023). The Effects of Frame Rate Variation on Game Player Quality of Experience. I *CHI '23: CHI Conference on Human Factors in Computing Systems*. New York NY, USA 23-28 april 2023, s. 1–10.

doi: 10.1145/3544548.3580665

Maung, D., Shi, Y. & Crawfis, R. (2012). Procedural textures using tilings with Perlin Noise, 2012 17th International Conference on Computer Games (CGAMES), Louisville, KY, USA 30 juli - 1 augusti 2012, s. 60-65.

doi: 10.1109/CGames.2012.6314553

Microsoft (2026). *Stopwatch Class*.

<https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-9.0> [2026-02-17]

Minecraft (2009) [Spel]. Dator. Stockholm: Mojang.

Odermatt, M., Marcilio, D. & Furia, C. A. (2022). Static Analysis Warnings and Automatic Fixing: A Replication for C# Projects. I *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA 15-

18 mars 2022, s. 805-816.

doi: 10.1109/SANER53432.2022.00098

Olano, M. (2005). Modified noise for evaluation on graphics hardware. I *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Los Angeles CA, USA 30-31 juli 2005, s. 105-110.

doi: 10.1145/1071866.1071883

Pasbanigoloojeh, R., Lawal, A., Daneshvar, B. & Lawal, A. (2026). Anisotropic Perlin Noise: Directional and Radial Pattern Generation for Texture Synthesis. I *2025 IEEE 17th International Conference on Computational Intelligence and Communication Networks (CICN)*. Goa, Indien 20-21 december 2025, s. 1284-1288.

doi: 10.1109/CICN67655.2025.11368098

Peck, J. (2016). FastNoise_CSharp. [Git Repository] Commit: f6b5b9d.

https://github.com/Auburn/FastNoise_CSharp [2026-02-17]

Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3), s. 287-296.

doi: 10.1145/566654.566636

Perlin, K. (2001). *Noise hardware*.

<https://userpages.cs.umbc.edu/olano/s2002c36/cho2.pdf> [2026-03-04]

Perlin, K. (2002). Improving noise. *ACM Transactions on Graphics (TOG)*, 21(3), s. 681-682.

doi: 10.1145/566654.566636

Rombauts, S. (2014). SimplexNoise. [Git Repository] Commit: 97e62c5.

<https://github.com/SRombauts/SimplexNoise> [2026-02-17]

Silva, K. P., Arcaro, L. F. & de Oliveira, R. S. (2020). Methods for Comparing Execution Times of Different Input Data when Real-Time Tasks Run on Complex Computer Architectures. I *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*. Florianopolis, Brasilien 24-27 november 2020, s. 65-73.

doi: 10.1109/SBESC51047.2020.9277839

Strand, A. & Prestberg, E. (2025). *PROCEDURELL GENERERING AV TERRÄNG - Hur skiljer beräkningstiden mellan Perlin och Simplex Noise vid terränggenerering?* Kandidatuppsats, Informationsteknologi. Högskolan i Skövde.

<https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Ahis%3Adiva-25057>

Takahashi, K. (2015). *PerlinNoise*. [Git Repository] Commit: b01ea8f.

<https://github.com/keijiro/PerlinNoise/blob/master/Assets/Perlin.cs> [2026-02-24]

Tan, C. E., Tan, W. H., Shamsudin, F. S., Navaratnam, S. & Ng, Y. Y. (2022).

Investigating the Impact of Latency in Mobile-Based Multiplayer Online Battle Arena (MOBA) Games. *International Journal of Creative Multimedia*, 3(1), s. 1-16.

doi: 10.33093/ijcm.2022.3.1.1

Terraria (2011) [Spel]. Dator. Floyds Knobs: Re-Logic.

Unity Technologies (2025). *Ultimate guide to profiling Unity games (Unity 6 edition)*. <https://unity.com/resources/ultimate-guide-to-profiling-unity-games-unity-6> [2026-02-28]

Unity Technologies (2026a). Manual: *Burst Compiler*. <https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html> [2026-02-26]

Unity Technologies (2026b). Manual: *Collect performance data introduction*. <https://docs.unity3d.com/6000.0/Documentation/Manual/profiling-collect-data-introduction.html> [2026-03-04]

Unity Technologies (2026c). Scripting API: *Time.realtimeSinceStartupAsDouble*. <https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Time-realtimeSinceStartupAsDouble.html> [2026-03-01]

Vitacion, R. J. & Liu, L. (2019). Procedural Generation of 3D Planetary-Scale Terrains. 2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT). Pasadena CA, USA 30 juli - 1 augusti, s. 70–77. doi: 10.1109/SMC-IT.2019.00014.

Wang, G., Su, S., Qin, B. & Wang, X. (2025). Dynamic volumetric cloud modelling with edge details refinement, *Computers & Graphics*, 133, article 104456. doi: 10.1016/j.cag.2025.104456

Zhang, S., Xu, Z., Liu, Z. & Liu, M. (2025). Noise Algorithms in Game Terrain Generation, *2025 10th International Conference on Image, Vision and Computing (ICIVC)*, Chengdu, Kina 16-18 juli 2025, s. 611-618. doi: 10.1109/ICIVC66358.2025.11200410