

GPU-ACCELERERAD VÄTSKESIMULERING MED SMOOTHED PARTICLE HYDRODYNAMICS

En validering av prestandavinster med compute shaders och en uniform grid för spatial hashing-baserad grannsökning

GPU-ACCELERATED FLUID SIMULATION USING SMOOTHED PARTICLE HYDRODYNAMICS

A validation of performance gains using compute shaders and a uniform grid for spatial hashing-based neighbour search

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2026

Effrosyni Moraitou

Handledare: Peter Sjöberg
Examinator: Mikael Thieme

Sammanfattning

Simulering av vatten med Smoothed Particle Hydrodynamics (SPH) är beräkningsintensivt men passar för interaktivt vatten i spel. Hrytsyshyn m.fl. (2023) visade att GPU-versionen av SPH gav en betydande prestandaökning, men studien saknade tillräcklig dokumentation för att möjliggöra reproducerbarhet. Denna studie undersöker om deras resultat kan reproduceras i en dokumenterad implementering genom att jämföra en CPU-version och en GPU-version av en identisk SPH-algoritm med Uniform Grid och spatial hashing som grannsökningsoptimering i spelmotorn Unity.

Kvasi-experimentet kördes på en bärbar dator med integrerad GPU och exekveringstiden mättes i millisekunder per tidssteg vid 1 000 till 75 000 partiklar. CPU-versionen överskred gränsen på 16,6 ms vid 5 000 partiklar, medan GPU-versionen höll sig under samma gräns upp till 10 000 partiklar. Vid 75 000 partiklar uppmättes en speedup-faktor på 16,55. Som framtida arbete föreslås tillämpning av GPU-sortering samt utvärdering av implementeringen på en dator med dedikerad GPU.

Nyckelord: SPH, GPU, vattensimulering, compute shaders, Unity

Innehållsförteckning

1	Introduktion	1
2	Bakgrund	2
2.1	Vatten i moderna spel	3
2.2	Smoothed Particle Hydrodynamics (SPH)	4
2.2.1	Utjämningskärnor (Smoothing Kernels)	5
2.2.2	Tryckkraft och Viskositet	6
2.3	Grafikprocessor (GPU)	7
2.4	Uniform Grid med spatial hashing	7
2.5	Hrytsyshyn m.fl. (2023) och reproducerbarhet	8
3	Problemformulering	9
3.1	Metodbeskrivning	9
3.2	Implementering	10
3.3	Metoddiskussion	11
3.4	Forskningsetik	11
4	Genomförande	13
4.1	CPU-implementering av SPH	13
4.1.1	Partikelinitialisering och rendering	13
4.1.2	SPH-kärnfunktioner	14
4.1.3	Kollisionshantering	15
4.2	GPU-implementeringen	15
4.2.1	Arkitektur och buffertar	15
4.2.2	Tidsstegshantering	16
4.2.3	Uniform Grid med spatial hashing	16
4.3	Prestandamätning	16
5	Resultat	18
5.1	CPU-implementering	18
5.2	GPU-implementering	18
5.3	Jämförelse CPU och GPU	19
5.4	Analys av resultat	21
6	Sammanfattning och diskussion	23
6.1	Sammanfattning	23
6.2	Diskussion	23
6.2.1	Resultatets trovärdighet och relation till tidigare forskning	23
6.2.2	Generaliserbarhet	24

6.3	Samhälleliga och etiska aspekter	25
6.4	Framtida arbete.....	26
	Referenser.....	27

1 Introduktion

Vattensimulering är en viktig del i modern spelutveckling. Att skapa realistiskt vatten kräver en balans mellan visuell realism och prestanda, eftersom datorspel, till skillnad från filmer, utför alla beräkningar i realtid. Lagrangian-, Eulerian- och hybridmetoder är de tre kategorierna för simulering av vatten. Valet beror på om vattnet ska interagera med spelare och objekt eller enbart behöver se visuellt realistiskt ut. Smoothed Particle Hydrodynamics (SPH) är en populär Lagrangian-metod som lämpar sig för interaktiva vattenscenarier, men är beräkningsintensiv.

Grafikprocessorn (GPU) erbjuder en möjlig lösning på prestandaproblemet eftersom SPH-algoritmen kan parallelliseras. Hrytsyshyn m.fl. (2023) fann att deras GPU-version kunde uppdatera 75 000 partiklar på cirka 10 millisekunder, medan deras CPU-version tog runt 690 millisekunder. Deras studie saknar dock tillräcklig dokumentation för att resultaten skulle kunna valideras eller jämföras på ett tillförlitligt sätt. Hrytsyshyn m.fl. (2023) nämner inte vilken hårdvara, utjämningskärnor (eng. *smoothing kernels*), verktyg eller optimeringar som användes i studien. Utan denna information är det svårt att avgöra om prestandaskillnaden beror på GPU-versionen eller på andra implementeringsval, såsom specifika optimeringar.

Syftet med denna studie är att undersöka om prestandaförbättringen för GPU-versionen av SPH även gäller för en dokumenterad och reproducerbar implementering. Frågeställningen är hur prestandan skiljer sig mellan en CPU-baserad och en GPU-baserad implementering av en identisk SPH-algoritm med Uniform Grid och spatial hashing som grannsökningsoptimering. Resultaten jämförs med de prestandaskillnader som redovisades av Hrytsyshyn m.fl. (2023). Spelmotorn Unity används som implementeringsmiljö eftersom den är en ofta använd plattform inom spelutveckling och erbjuder stöd för compute shaders.

Metoden som valdes är ett kvantitativt kvasi-experiment, där en CPU-version och en GPU-version av samma SPH-algoritm implementeras och jämförs under identiska villkor på samma dator. Exekveringstiden mättes i millisekunder per tidssteg för partikelantal mellan 1 000 och 75 000 och jämförelsegränsen sattes till 16,6 millisekunder, vilket motsvarar 60 bilder per sekund. CPU-versionen implementeras i C# och GPU-versionen i HLSL som compute shaders. Båda versionerna använder Poly6-utjämningskärnan för densitetsberäkningar, Spiky-gradienten för tryckkraftsberäkningar och viskositets-laplacian för viskositetsberäkningar.

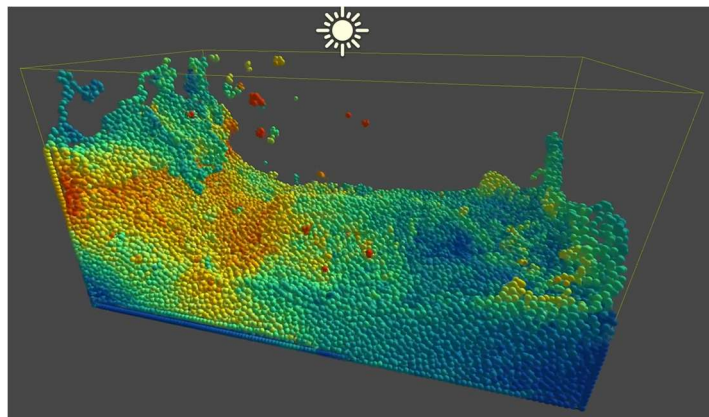
Resultaten visar att GPU-versionen ger en speedup-faktor på 16,55 vid 75 000 partiklar jämfört med CPU-versionen, vilket stämmer överens med resultaten från Hrytsyshyn m.fl. (2023) om GPU:ns fördelar. GPU-versionen låg under jämförelsegränsen på 16,6 ms upp till 10 000 partiklar, medan CPU-versionen överskred samma gräns redan vid 5 000 partiklar. Kvasi-experimentet kördes på en bärbar dator med integrerad GPU, vilket begränsade prestandan men påverkade inte den relativa jämförelsen mellan implementeringarna.

2 Bakgrund

En viktig del av modern spelutveckling är skapandet av övertygande och atmosfäriska miljöer, vilket ställer höga krav på simulering och rendering av vatten. Vattensimulering inom spelutveckling är särskilt utmanande på grund av kravet på balans mellan visuell realism och realtidsprestanda (Huang & Yi 2024). Till skillnad från filmer, där prestandakrävande simuleringar kan förrenderas bild för bild under en lång tid, behöver datorspel kunna köra dessa simuleringar i realtid. Utöver detta behöver simuleringen använda en begränsad andel av processorkraften, eftersom den återstående beräkningskraften används till övriga delar av datorspelet.

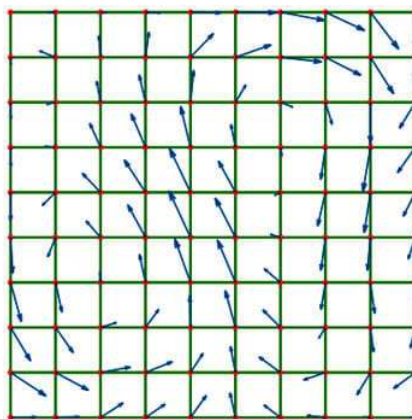
Det finns flera olika metoder för simulering av vatten och dessa metoder är Lagrangian-, Eulerian- och hybridmetoder (Bridson & Müller-Fischer 2007). Lagrangian-metoden behandlar vattnet som om det vore ett partikelsystem. Detta kan liknas vid att följa enskilda vattendroppar och spåra hur var och en av dem rör sig. Eulerian-metoden delar upp vattnet i ett rutnät av celler, som ett schackbräde (Bridson & Müller-Fischer 2007). I stället för att följa enskilda vattendroppar, som i Lagrangian-metoden, mäts vattnets egenskaper, som hastighet och densitet, i varje cell vid varje tidssteg. Hybridmetoder kombinerar styrkorna hos de två metoderna på olika sätt.

I Figur 1 nedan illustreras grundprinciper i Lagrangian-metoden, där partiklarna rör sig fritt.



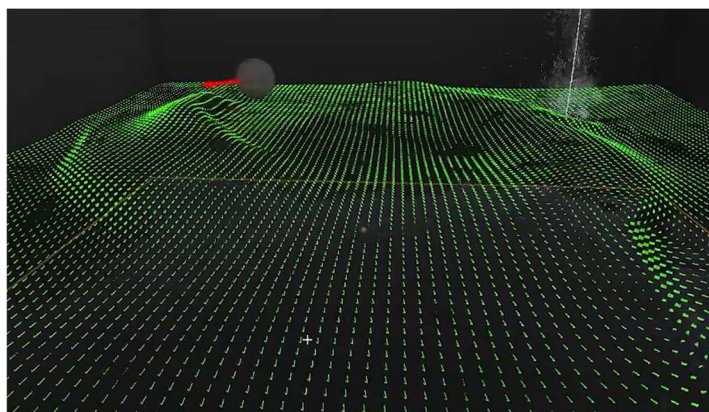
Figur 1 visar hur partiklarna rör sig fritt i Lagrangian-metoden (Laque 2023).

Figur 2 nedan visar motsvarande grundprincip för Eulerian-metoden, där vattnets rörelse i stället mäts i ett rutnät av lika stora celler.



Figur 2 visar hur Eulerian-metoden mäter vattnets rörelse i fasta celler i ett rutnät och visar dess riktning med hjälp av pilarna (West 2008).

Figur 3 visar ett exempel på hur Eulerian-metoden tillämpas i spelet Still Wakes the Deep.

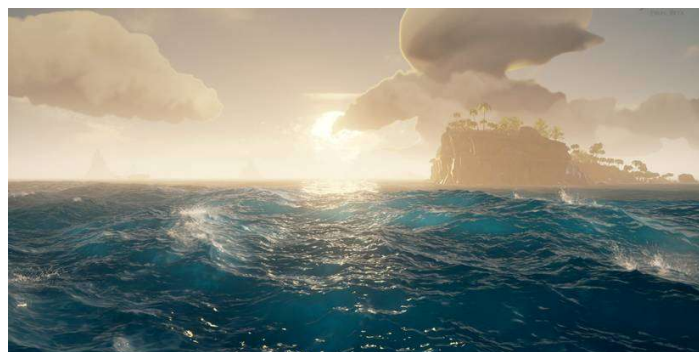


Figur 3 visar ett exempel på hur Eulerian-metoden används i spelet Still Wakes the Deep, där vattenytan simulerades med hjälp av ett rutnät (Wheater 2024).

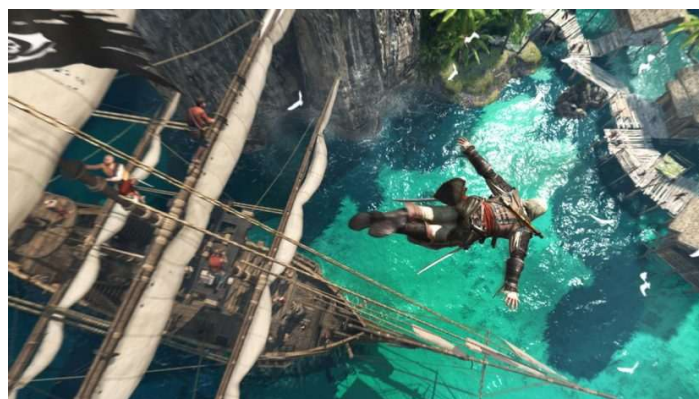
2.1 Vatten i moderna spel

I modern spelutveckling används olika metoder för simulering av vatten. Valet av metod beror på målet för realism, prestanda och interaktivitet. I de flesta fall används spelutvecklare metoder som ger illusionen av realistiskt vatten, eftersom fysiskt korrekta vattensimuleringar är för beräkningsintensiva för realtidsprogram. I spel som Sea of Thieves (2018), Assassin's Creed IV: Black Flag (2013) och Still Wakes the Deep (2024) är havet en central del av spelupplevelsen.

Figur 4-6 illustrerar havets utseende i respektive spel.



Figur 4: Havet i Sea of Thieves (Natividad 2022).



Figur 5: Havet i AC IV: Black Flag (Failes 2013).



Figur 6: Havet i Still Wakes the Deep (nass3x).

Dessa spel ställer olika krav på metoden för simuleringen av vatten, eftersom kraven beror på om vattnet behöver interagera med spelare och objekt, eller om det enbart behöver se visuellt realistiskt ut. I dessa spel används huvudsakligen Eulerian-baserade metoder eller hybridmetoder för att simulera stora vattenytor som hav. Dessa metoder är effektiva för att skapa ett visuellt övertygande hav utan att beräkningsbelastningen blir alltför hög. När vatten däremot behöver interagera med spelare och objekt, exempelvis när en karaktär simmar eller ett föremål landar i vattnet, används ofta Lagrangian-baserade metoder som Smoothed Particle Hydrodynamics (SPH), eftersom dessa är bättre lämpade för att hantera sådana interaktioner. Eulerian- och hybridmetoder kan dock också användas för att uppnå liknande beteende.

2.2 Smoothed Particle Hydrodynamics (SPH)

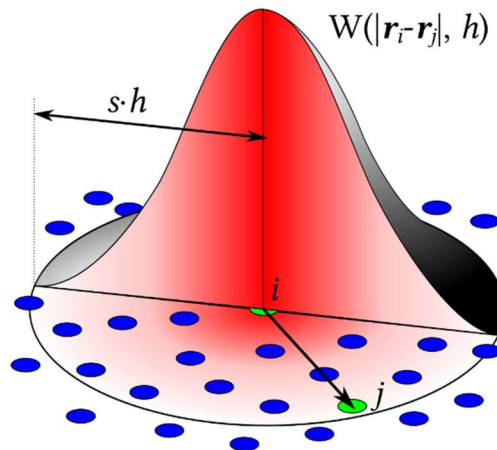
Smoothed Particle Hydrodynamics (SPH) är en populär Lagrangian-metod för vätskesimuleringar, specifikt när vätskan behöver interagera med olika objekt. SPH uppfanns av Lucy (1977) samt Gingold och Monaghan (1977) för astrofysiksimuleringar, men anpassades senare även för användning inom datorgrafik.

Idén i SPH är att vätskan representeras av ett stort antal partiklar, där varje partikel har egenskaper som massa, hastighet och densitet. Varje partikel påverkas av sina grannpartiklar inom ett visst avstånd h , som kallas utjämningskärnans (eng. *smoothing kernel*) radie. En partikel kan föreställas vara omgiven av en osynlig cirkel. Alla partiklar inom den cirkeln bidrar till bestämningen av tryck och densitet hos partikeln i centrum.

I varje tidssteg utför SPH-algoritmen följande steg:

1. Hitta grannpartiklar inom radien h för varje partikel
2. Beräkna externa krafter (gravitation, användarinteraktioner)
3. Beräkna densitet för varje partikel baserat på grannarna
4. Beräkna tryck utifrån densiteten
5. Beräkna tryckkraft och viskositet
6. Uppdatera varje partikels hastighet och position baserat på ackumulerade krafter.

Figur 7 nedan illustrerar hur en partikel påverkas av sina grannpartiklar inom utjämningsradien h och hur påverkan minskar med avståndet.



Figur 7: En partikel i (färgad grön) omgiven av en cirkel med radien h . Inuti cirkeln finns grannpartiklar. Den röda färgen illustrerar partikelns påverkan vid beräkningar av densitet och tryckkraft, och hur den påverkan minskar med avståndet (Wikipedia 2026).

Den matematiska grunden för dessa steg beskrivs av följande ekvationer. Enligt Waseem och Hong (2025) är SPH en interpolationsmetod för partikelsystem där varje partikel har fysiska egenskaper som massa, hastighet och densitet. Den första ekvationen beskriver hur värdet av vilken kvantitet f som helst kan beräknas vid positionen \mathbf{r}_a :

$$f(\mathbf{r}_a) = \sum_b m_b \frac{f(\mathbf{r}_b)}{\rho_b} W(\mathbf{r}_a - \mathbf{r}_b, h) \quad (1)$$

där indexet b representerar grannpartiklar inom en utjämningskärnas radie h , m är massan, ρ är densiteten och funktionen $W(\mathbf{r}_a - \mathbf{r}_b, h)$ är en utjämningskärna med en radie h (Hrytsyshyn m.fl. 2023). I Ekvation 1 har varje partikel med indexet b egenskaper såsom massa och densitet. Summeringen innebär att värdet för en partikel a påverkas av grannpartiklar inom en utjämningskärnas radie.

2.2.1 Utjämningskärnor (Smoothing Kernels)

Olika beräkningar i SPH kräver olika typer av utjämningskärnor, på samma sätt som olika verktyg behövs för olika jobb. En utjämningskärna som fungerar bra för densitetsberäkningar kan ge felaktiga eller instabila resultat vid exempelvis beräkning av tryckkrafter. Av denna anledning används ofta olika utjämningskärnor för olika storheter i SPH-simuleringar.

Enligt Hrytsyshyn m.fl. (2023) fungerar utjämningskärnan $W(\mathbf{r}_a - \mathbf{r}_b, h)$ som en vikt. Den bestämmer påverkan av grannpartikeln b på partikeln a , baserat på avståndet mellan dem och längden h . Valet av utjämningskärna är avgörande för SPH-simuleringens stabilitet och beräkningseffektivitet, eftersom dessa egenskaper varierar mellan olika utjämningskärnor (Hrytsyshyn m.fl. 2023). En funktion kan kallas en utjämningskärna endast om den uppfyller specifika egenskaper. Dessa egenskaper beskrivs av Liu och Liu (2010), där de även beskriver hur en egen utjämningskärna kan skapas.

För beräkningen av en vätskas tryckkrafter och viskositet i SPH används utjämningskärnors derivator, vilka beskriver hur utjämningskärnans värde förändras med avståndet. Gradienten (∇W) är den första derivatan av en utjämningskärna och beskriver hur snabbt utjämningskärnans värde förändras och i vilken riktning. Gradienten tar en funktion eller skalär och returnerar en vektor. Laplacian ($\nabla \cdot \nabla, \nabla^2$ eller Δ) är den andra derivatan av en utjämningskärna och beskriver hur denna förändring i sig förändras, likt skillnaden mellan hastighet och acceleration, laplacian tar en vektor och returnerar en skalär (Christian & Laursen 2013).

Poly6-utjämningskärnan används ofta för densitetsberäkningar eftersom den är stabil och beräkningsmässigt effektiv, något som även Waseem och Hong (2025) tillämpar. Poly6 definieras enligt följande:

$$W_{\text{poly6}}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

För beräkningen av tryckkrafter använde Müller, Charypar och Gross (2003) utjämningskärnan Spiky, specifikt dess gradient. Spiky valdes i stället för Poly6 eftersom Poly6 tenderade att bilda partikelkluster vid beräkning av tryckkrafter, medan Spiky-gradienten undvek klusterbildning. Spiky definieras som:

$$W_{\text{spiky}}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Enligt Müller, Charypar och Gross (2003) kräver viskositetsberäkningen mer specifika utjämningskärnor. Om en vanlig utjämningskärna används vid grovt samplade hastighetsfält kan kärnans Laplacian ge negativa värden, vilket resulterar i att partiklarnas relativa hastighet ökar i stället för att dämpas. Av denna anledning Müller, Charypar och Gross (2003) utvecklade denna utjämningskärna:

$$W_{\text{viscosity}}(\mathbf{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

2.2.2 Tryckkraft och Viskositet

I denna studies simulering är massan av en partikel konstant medan densiteten ρ_b varierar, vilket betyder att den behöver beräknas vid varje tidssteg. Densiteten av en partikel beräknas med denna ekvation:

$$\rho(\mathbf{r}_a) = \sum_b m_b \frac{\rho_b}{\rho_b} W(r_a - r_b, h) = \sum_b m_b W(r_a - r_b, h) \quad (5)$$

Efter beräkningen av densiteten kan trycket beräknas. Enligt Müller, Charypar och Gross (2003) kan trycket beräknas enligt en modifierad version av den ideala gaslagen:

$$p = k(\rho - \rho_0) \quad (6)$$

där p är trycket, k är en gaskonstant som varierar beroende på temperaturen, ρ är densitet och ρ_0 är vilodensiteten.

Navier-Stokes ekvationer är fysikens grundläggande ekvationer för hur vätskor beter sig, på samma sätt som Newtons lagar beskriver krafter och rörelse. Innan tryckkraften och viskositeten beräknas introduceras Navier-Stokes ekvationer:

$$\rho \left(\frac{\partial v}{\partial t} + v \cdot \nabla v \right) = -\nabla p + \rho g + \mu \nabla^2 v \quad (7)$$

där $-\nabla p$ är tryckkraften, g är en extern kraft som gravitation, μ är vätskans viskositetskonstant och $\nabla^2 v$ är viskositeten. Navier-Stokes ekvationer är nödvändiga eftersom de beskriver dynamiken hos vätskor. Denna studie behandlar inte härledningen av tryckkrafts- och viskositetsekvationerna från Navier-Stokes ekvationer.

InteractiveComputerGraphics (2025) och Müller, Charypar och Gross (2003) visar de härledda tryckkrafts- och viskositetsekvationerna:

$$f_a^{\text{tryckkraft}} = - \sum_b m_b \left(\frac{p_a}{\rho_a^2} + \frac{p_b}{\rho_b^2} \right) \nabla W(r_a - r_b, h) \quad (8)$$

$$f_a^{\text{viskositet}} = \mu \sum_b m_b \frac{p_b - p_a}{\rho_b} \nabla^2 W(r_a - r_b, h) \quad (9)$$

Den totala kraften på en partikel enligt Navier-Stokes ekvationer är summan av tryckkraften, viskositeten och den externa kraften.

2.3 Grafikprocessor (GPU)

Implementeringen av SPH på en grafikprocessor (GPU) är av särskilt intresse eftersom SPH-algoritmen kan parallelliseras (Ramakrishnan & McGraw 2023). Varje partikels nya tillstånd kan beräknas oberoende av övriga partiklar, vilket möjliggör en GPU-implementering av SPH. En CPU-implementering av SPH utför beräkningarna sekventiellt eller med begränsad parallellisering, vilket begränsar prestandan när antalet partiklar ökar i simuleringen.

Compute shaders används för allmänna beräkningar på GPU:n. Waseem och Hong (2025) förklarar att compute shaders möjliggör acceleration av komplexa simuleringar som SPH, vilket ger stora prestandaökningar.

För applikationer som spel är denna hastighet nödvändig, eftersom en simulering som SPH behöver kunna köras i realtid. Kravet för en flytande spelupplevelse är 60 bilder per sekund, vilket är 16,6 millisekunder per bild (Ramakrishnan och McGraw 2023). Det bör dock noteras att kravet på 60 bilder per sekund (16,6 ms) inte är universellt inom spelindustrin. Vissa spel, som konsolspel, accepterar i stället 30 bilder per sekund (33,3 ms). Denna studie använder 16,6 millisekunder som realtidsgräns eftersom 60 bilder per sekund representerar standarden för datorspel och andra interaktiva applikationer.

2.4 Uniform Grid med spatial hashing

En stor utmaning i SPH-simuleringar är grannsökning, det vill säga hur varje partikel effektivt kan hitta sina grannpartiklar inom utjämningsradien h . Utan optimering krävs att varje partikel jämförs med alla andra partiklar, vilket ger en tidskomplexitet på $O(n^2)$, något som snabbt gör simuleringen oanvändbar vid höga partikelantal.

Green (2010) introducerade Uniform Grid med spatial hashing som en effektiv lösning på grannsökningproblemet. Metoden delar upp simuleringsutrymmet i ett rutnät av lika stora celler, där varje partikel tilldelas en cell baserat på dess position. Cellkoordinaterna omvandlas till ett hashvärde via primtalsmultiplikation, vilket gör det möjligt att lagra och söka i en array utan att rutnätets dimensioner behöver vara kända i förväg (Lague 2023). Vid grannsökning behöver därmed bara de 27 närmaste cellerna ($3 \times 3 \times 3$) sökas igenom i stället för alla partiklar, vilket minskar antalet jämförelser från $O(n^2)$ till ungefär $O(n \log n)$.

2.5 Hrytsyshyn m.fl. (2023) och reproducerbarhet

Studien av Hrytsyshyn m.fl. (2023) visar en stor prestandaförbättring vid implementering av SPH med compute shaders. Implementering av Hrytsyshyn m.fl. (2023) kan uppdatera 75 000 partiklar på cirka 10 millisekunder jämfört med CPU-implementeringen som tog omkring 690 millisekunder. Denna studies val av 16,6 millisekunder som realtidsgräns motiveras av att denna gräns representerar standardgränsen för realtid i datorspel, men också för att det möjliggör jämförelse med Hrytsyshyn m.fl. (2023). Studien ger dock inte tillräckligt med detaljer för att kunna reproducera eller jämföra resultaten på ett tillförlitligt sätt. Följande information saknas:

- Vilka verktyg eller vilken miljö som användes för implementeringen, exempelvis om SPH-algoritmen implementerades i OpenGL, Vulkan, CUDA eller en spelmotor som Unity eller Unreal.
- Vilka utjämningskärnor samt deras gradienter och laplacians som användes i simuleringen.
- Vilken hårdvara simuleringen kördes på, exempelvis om den kördes på en bärbar dator eller en högpresterande dator.
- Vilka optimeringar som tillämpades utöver compute shaders för att uppnå cirka 10 millisekunder för 75 000 partiklar.

Utan dessa detaljer är det svårt att avgöra om de nämnda prestandaskillnaderna beror på GPU-implementeringen eller på specifika implementeringsval som grannsökningsoptimeringar eller val av utjämningskärnor.

Hrytsyshyn m.fl. (2023) nämner grannsökning som en del av sin implementering men förklarar inte vilken optimeringsmetod som används, om sådan användes. I denna studie används Uniform Grid med spatial hashing, se avsnitt 2.4, som grannsökningsoptimering, vilket dokumenteras för att möjliggöra reproducerbarhet.

Spelmotorn Unity är en ofta använd plattform för spelutveckling och erbjuder stöd för compute shaders, vilket gör den till en lämplig miljö för SPH-implementeringen i denna studie.

3 Problemformulering

Realtidssimulering av vatten förblir en teknisk utmaning inom spelutveckling. Strävan efter visuell realism behöver balanseras mot realtidsprestanda eftersom datorer har en begränsad mängd resurser som också ska gå till andra delar av ett spel såsom fiendelogik och spelarlogik. När vatten behöver interagera med spelare och andra objekt är SPH en passande metod (se avsnitt 2.2), men metoden är beräkningsintensiv. GPU-parallellisering via compute shaders är en potentiell lösning på detta beräkningsproblem (se avsnitt 2.3). För att minska beräkningskostnaden ytterligare används Uniform Grid med spatial hashing (se avsnitt 2.4). Hrytsyshyn m.fl. (2023) visar att en GPU-implementering av SPH ger stora prestandaförbättringar, men studien saknar tillräcklig information för att reproducera arbetet (se avsnitt 2.5).

Frågeställningen är: Hur skiljer sig prestandan mellan en CPU-baserad och en GPU-baserad (compute shaders) implementering av en identisk SPH-algoritm med Uniform Grid och spatial hashing som grannsökningsoptimering när de körs, jämfört med de prestandaskillnader som rapporterades av Hrytsyshyn m.fl. (2023)?

Denna studie är relevant ur både ett akademiskt och ett praktiskt perspektiv. Ur det akademiska perspektivet saknar studien av Hrytsyshyn m.fl. (2023) tillräcklig information för att resultaten ska kunna valideras av andra forskare. Denna studie bidrar med en definierad och dokumenterad SPH-implementering i Unity. I implementeringen specificeras utjämningskärnor, Uniform Grid med spatial hashing som grannsökningsoptimering samt hårdvara, vilket möjliggör validering.

Ur det praktiska perspektivet är resultaten viktiga för spelutvecklare som vill använda SPH i sina spel. Simulering av vatten med SPH är beräkningsintensivt och implementeringsvalet kan ha stor påverkan på spelets prestanda. Den tydliga jämförelsen mellan CPU- och GPU-implementeringarna i Unity ger spelutvecklare en klar grund för att avgöra om SPH passar för deras specifika mål, exempelvis när vatten behöver interagera med spelare och objekt i realtid.

3.1 Metodbeskrivning

Metoden som valdes för genomförandet av denna studie var ett kvantitativt kvasi-experiment för att validera en tidigare studie. Som beskrivs i avsnitt 2.5, saknar Hrytsyshyn m.fl. (2023) tillräcklig dokumentation för reproducerbarhet, vilket motiverade valet av kvasi-experimentet som metod för denna studie.

Kvasi-experimentet genomfördes i en kontrollerad miljö, det vill säga vid samma dator, för att minska påverkan från faktorer som datorns hårdvara. Datorn hade följande hårdvara: 12th Gen Intel(R) Core(TM) i3-1215U, Intel(R) UHD Graphics (integrerad grafikprocessor) med 128 MB dedikerat minne och 8 GB RAM. En integrerad GPU delar systemminne med CPU:n, vilket kan påverka GPU-versionens prestanda jämfört med en dedikerad GPU.

Kvasi-experimentet valdes eftersom typen av data som metoden genererade ansågs passa bäst för att besvara frågeställningen. Enligt Borg & Westerlund (2014, s. 14) kännetecknas ett kvasi-experiment av att det finns kontroll men ingen randomisering. Detta kvasi-experiment involverar inga människor. Eftersom de oberoende variablerna kontrollerades klassificeras studien som ett kvasi-experiment. Studiens metod räknas inte som ett äkta experiment eller icke-experiment, eftersom äkta experiment kräver både kontroll och randomisering, medan icke-experiment saknar kontroll och randomisering.

CPU-versionen implementerades i standard C# utan användning av Unitys Burst Compiler, Jobs System eller DOTs. Dessa verktyg kunde ge stora prestandaförbättringar för beräkningsintensiva algoritmer som SPH. Eftersom dessa optimeringar inte användes visade CPU-versionen inte sin fulla potential, vilket påverkade speedup-faktorn. Studien mätte skillnaden mellan en ej optimerad CPU-version och en GPU-version. En optimerad CPU-version med Burst och Jobs hade troligen minskat speedup-faktorn och detta bör beaktas vid tolkning av resultaten.

Exekveringstiden för CPU-versionen mäts med hjälp av System.Diagnostics.Stopwatch i C#. Tidtagningen startar precis innan SPH-algoritmen börjar och stoppas direkt efter att alla beräkningar för ett tidssteg har slutförts. För GPU-versionen används AsyncGPUReadback för att säkerställa att GPU:n har slutfört alla beräkningar innan tiden stoppas. Detta beror på att GPU-beräkningarna är asynkrona och inte nödvändigtvis klara när CPU:n fortsätter sitt arbete. Båda mätningarna räknar tiden i millisekunder per tidssteg.

Kvasi-experimentet körs i Unity utan någon rendering av partiklarna för att isolera beräkningstiden för SPH-algoritmen från renderingstiden. Detta är relevant eftersom datorspel delar processorkraft mellan simulering och rendering, medan denna studies syfte är att mäta enbart simuleringens prestanda. CPU-versionen av simuleringen körs i Unitys Update-loop och GPU-versionen anropar compute shaders varje bildruta, vilket motsvarar hur SPH används i spel.

För att undersöka hur prestandan förändras med problemstorleken körs kvasi-experimentet med varierande antal partiklar. Partikelantalet varieras i stegen 1 000, 5 000, 10 000, 25 000, 50 000 och 75 000 partiklar. Det högsta antalet partiklar motsvarar det partikelantal som används av Hrytsyshyn m.fl. (2023). Detta gör det möjligt att observera vid vilket partikelantal CPU-versionen blir oanvändbar för en realtidsapplikation som ett spel, samt hur GPU-versionen förändras i jämförelse.

Det primära måttet som används är genomsnittlig exekveringstid i millisekunder per tidssteg. För varje partikelantal körs simuleringen i 500 tidssteg, där de första 50 tidsstegen exkluderas för att undvika uppvärmningseffekter. Med uppvärmningseffekt menas den initiala period då systemet, som CPU:n och GPU:n, ännu inte har stabiliserat sig och kan ge missvisande mätvärden. Medeltiden beräknas sedan över de återstående 450 tidsstegen. Gränsen för realtidsprestanda i ett datorspel sätts till 16,6 millisekunder, vilket motsvarar 60 bilder per sekund (Ramakrishnan och McGraw 2023).

3.2 Implementering

Samma SPH-algoritm implementeras i två versioner:

- CPU-versionen implementeras i C# och exekveras i Unitys Update-loop.
- GPU-versionen implementeras i HLSL som en compute shader, där data hanteras via Unitys ComputeBuffer.

Båda versionerna använder samma fysikekvationer för densitets-, tryckkrafts- och viskositetsberäkningarna samt samma Uniform Grid med spatial hashing optimering för grannsökning. Först implementeras en naiv CPU-baserad SPH-simulering. Därefter implementeras samma algoritm på GPU för att verifiera att båda simuleringarna fungerar korrekt. Slutligen implementeras Uniform Grid med spatial hashing som grannsökningsoptimering, först för CPU-versionen och sedan för GPU-versionen.

För att säkerställa att CPU-versionen och GPU-versionen skapar korrekta och identiska resultat genomförs visuell inspektion och numerisk verifiering av båda simuleringarna. Partiklarna visar samma övergripande beteende i båda versionerna, exempelvis att de sjunker mot botten på grund av gravitation och att de inte överlappar varandra på grund av tryckkrafter. Vid numerisk verifiering jämförs densitets- och tryckvärden för ett antal utvalda partiklar mellan CPU-versionen och GPU-versionen för att bekräfta att beräkningarna ger samma resultat med en acceptabel felmarginal. Med acceptabel felmarginal menas en avvikelse på mindre än 0,01% mellan CPU- och GPU-versionernas beräknade värden.

3.3 Metoddiskussion

Kvasi-experiment väljs eftersom Borg & Westerlund (2014, s. 11–12) beskriver hur experiment är den bästa metoden för att undersöka orsak och verkan, det vill säga att forskaren kan manipulera den oberoende variabeln och se effekten av detta på den beroende variabeln. I denna studie manipuleras vilken implementering av SPH (CPU/GPU) som körs och antalet partiklar för mätning av effekten på beräkningstiden.

Studien är ett kvasi-experiment i stället för ett äkta experiment eftersom implementeringarna är förutbestämda. Implementeringarna är en CPU-version och en GPU-version. De förutbestämda implementeringarna gör att randomisering inte är möjlig eller meningsfull i denna studie. Fördelen med kvasi-experimentet är att alla variabler mellan CPU-versionen och GPU-versionen är lika och att mätningarna upprepas, vilket ökar reliabiliteten (Eliasson 2019, s. 14) och validiteten hos resultaten enligt Eliasson (2019, s. 16).

Enligt Alklind Taylor (2023) kan automatisering användas för att spara data i en logg, vilket hade kunnat kombineras med kvasi-experimentet i denna studie. Mätrutinen var automatiserad på så sätt att medelvärde, minimum och maximum beräknades och skrevs automatiskt i Unitys konsol efter 500 tidssteg. Värdena lästes sedan av manuellt från konsolens "Debug.Logs" för varje partikelantal. En automatisk export till en extern loggfil implementerades inte eftersom det hade krävt ytterligare utvecklingstid. I efterhand hade det troligen sparat tid, särskilt vid de upprepade körningarna med olika partikelantal.

Utöver prestandamätningar är estetiska aspekter av vattensimuleringar också relevanta inom spelutveckling, exempelvis hur vattnet ser ut för spelare. För att undersöka detta hade enkäter enligt Alklind Taylor (2023) varit en lämplig metod, där deltagare skulle bedöma hur realistiskt vattnet ser ut i de olika implementeringarna. Denna studie fokuserar dock enbart på prestanda och inkluderade därför inte sådana mätningar, men detta är ett intressant område för framtida forskning.

3.4 Forskningsetik

Detta examensarbete omfattade inte människor eller djur, vilket innebar att de fyra huvudkraven från Vetenskapsrådet (2002) inte var tillämpliga. Detta arbete bedrevs i enlighet med de grundläggande principerna för god forskningssed (Vetenskapsrådet 2024). Enligt Vetenskapsrådet (2024) är dessa principer tillförlitlighet, ärlighet och ansvar.

För att säkerställa simuleringens kvalitet lades stor vikt vid planeringen av kvasi-experimentet så att mätningarna av prestandan mellan CPU-versionen och GPU-versionen skulle vara rättvisa. Implementeringen av SPH-algoritmen och Uniform Grid med spatial hashing optimeringen beskrevs noggrant, vilket gav läsaren möjligheten att bedöma resultatens tillförlitlighet och kvalitet.

I denna studie togs ansvar för hela studiens process och ärlighet eftersträvades i redovisningen av resultaten. Källkoden dokumenterades noggrant och alla steg i utvecklingsprocessen redovisades korrekt för att möjliggöra validering.

4 Genomförande

Detta kapitel beskriver implementeringen av de två SPH-versionerna som används i denna studie. Avsnitt 4.1 förklarar CPU-implementeringen som kodades i C# samt partikelinitialisering, SPH-kärnfunktioner och kollisionshantering. Avsnitt 4.2 beskriver GPU-implementeringen med compute shaders, dess arkitektur, tidssteghanteringen och grannsökningsoptimering. Avsnitt 4.3 redogör för prestandamätningen som används i studien för att jämföra de två SPH-implementeringarna.

4.1 CPU-implementering av SPH

Implementeringen påbörjades med utvecklingen av den CPU-baserade SPH-simuleringen i Unity 6. SPH-implementeringen bygger på Müller, Charypar och Gross (2003). Müller m.fl. (2003) visar hur SPH-algoritmen kan implementeras för vätskor som vatten och presenterar de utjämningskärnor och formler som implementeringen bygger på.

Implementeringen krävde tre justeringar jämfört med originalstudien. För det första byttes tryckkraftsformeln ut mot den symmetriska WCSPH-varianten från InteractiveComputerGraphics (2025) för att undvika klusterbildning. För det andra ersattes Cubic Spline-utjämningskärnan med Ekvation 2 för densitetsberäkningar på grund av instabilitet vid partikelavstånd nära noll. För det tredje justerades partikelinitialiseringen och tidssteget för att simuleringen skulle hållas stabil (Tjernell och Andersson 2023). Dessa justeringar beskrivs i detalj i följande avsnitt.

Eftersom originalstudien av Hrytsyshyn m.fl. (2023) inte nämnde vilka utjämningskärnor eller formler som användes, kompletterades simuleringens implementering med studien av Müller, Charypar och Gross (2003), InteractiveComputerGraphics (2025) samt studien av Waseem och Hong (2025) för att täcka dessa informationsluckor. För den praktiska implementeringen användes Lague (2023) och AJTech (2023).

4.1.1 Partikelinitialisering och rendering

Pseudokoden nedan visar hur partiklarna initialiseras vid simuleringens början med position, hastighet, densitet och kraft satta till sina startvärden.

```
InitParticles(){
    avstånd = utjämningsradie * 0,95
    antalPerAxel = (antalPartiklar^(1/3))
    startStorlek = (antalPerAxel - 1) * avstånd
    startPos = behållarBotten - (startStorlek.x, 0, startStorlek.z) * 0,5
    i = 0
    for (x,y,z) = 0 till antalPerAxel gör
        position[i] = startPos + (x,y,z) * avstånd + jitter
        hastighet[i] = (0,0,0)
        densitet[i] = viloDensitet
        kraft[i] = (0,0,0) i++}
```

Partiklarna placerades i en kubisk form med ett avstånd på $0,95 * utj\u00e4mningsradien$. Faktorn $0,95$ valdes eftersom ett avstånd p\u00e5 $1,0 * utj\u00e4mningsradien$ orsakade explosiva tryckkrafter vid simuleringens start. Detta uppstod p\u00e5 grund av att partiklarna befann sig vid utj\u00e4mningsradiens gr\u00e4ns, vilket ledde till att densitetsber\u00e4kningarna blev instabila. Startpositionen f\u00f6rsk\u00f6ts mot beh\u00e5llarens botten ($boundarySize.y * 0,4$) f\u00f6r att simulera vattnets tillst\u00e5nd under gravitationens p\u00e5verkan i b\u00f6rjan av simuleringen.

F\u00f6r rendering valdes `Graphics.RenderMeshIndirect`, vilket m\u00f6jliggjorde direkt l\u00e4sning av data, s\u00e5som partikelpositioner och hastigheter, fr\u00e5n `ComputeBuffers` p\u00e5 GPU:n utan att data beh\u00f6vde \u00f6verf\u00f6ras tillbaka till CPU:n varje bildruta. Detta eliminerade en potentiell flaskhals som annars hade uppst\u00e5tt vid rendering av tusentals partiklar.

F\u00f6r visuell inspektion implementerades hastighetsbaserad f\u00e4rg\u00e4ndring av partiklarna via en Unlit shader. En Unlit shader valdes f\u00f6r att undvika belysningsber\u00e4kningar som inte var relevanta f\u00f6r simuleringen.

4.1.2 SPH-k\u00e4rnfunktioner

Densitetsber\u00e4kningen implementerades med Ekvation 2 enligt M\u00fcller, Charypar och Gross (2003). Denna ekvation valdes framf\u00f6r Cubic Spline-utj\u00e4mningsk\u00e4rnan. Cubic Spline-utj\u00e4mningsk\u00e4rnan orsakade i vissa fall division med noll och anv\u00e4ndes d\u00e4rf\u00f6r inte i den slutgiltiga implementeringen. Detta berodde sannolikt p\u00e5 att Cubic Spline inte hade samma nollgr\u00e4ns som Ekvation 2 vid avst\u00e5nd n\u00e4ra noll, vilket ledde till instabilitet vid t\u00e4tt packade partiklar.

Ekvation 2 hade dock en k\u00e4nd begr\u00e4nsning vid tryckkraftsber\u00e4kningar d\u00e4r partiklarna tenderade att bilda kluster under h\u00f6gt tryck. Detta berodde p\u00e5 att gradientv\u00e4rderna f\u00f6r Ekvation 2 n\u00e4rmade sig noll i centrum, vilket innebar att repulsionskrafterna f\u00f6rsvann n\u00e4r partiklarna var n\u00e4ra varandra (M\u00fcller, Charypar och Gross 2003). Av denna anledning anv\u00e4ndes Ekvation 2 enbart f\u00f6r densitetsber\u00e4kningar medan Ekvation 3, som inte hade samma problem som Ekvation 2, anv\u00e4ndes f\u00f6r tryckkraftsber\u00e4kningar. Trycket ber\u00e4knades med den ideala gaslagen, Ekvation 6, som togs fr\u00e5n samma studie.

F\u00f6r tryckkrafterna implementerades Ekvation 3, specifikt dess gradient. I b\u00f6rjan anv\u00e4ndes den tryckkraftsformel som presenteras av M\u00fcller, Charypar och Gross (2003), men den orsakade att partiklarna klumpades ihop i st\u00e4llet f\u00f6r att repellera varandra. Problemet l\u00f6stes med den symmetriska formeln fr\u00e5n Weakly Compressible SPH-varianten, Ekvation 8 (InteractiveComputerGraphics 2025). Den symmetriska formeln garanterade att tryckkrafterna mellan tv\u00e5 partiklar var lika stora men motsatt riktade, vilket bevarade linj\u00e4rt och vinklat moment. Dessutom f\u00f6rhindrade den att partikelklumpar bildades, ett problem som den ursprungliga formeln inte l\u00f6ste (InteractiveComputerGraphics 2025).

Viskositetskrafterna ber\u00e4knades enligt Ekvation 9 i M\u00fcller, Charypar och Gross (2003) med Ekvation 4 som utj\u00e4mningsk\u00e4rna, vilken valdes f\u00f6r att den var specifikt skapad f\u00f6r viskositetsber\u00e4kningar. Denna utj\u00e4mningsk\u00e4rna garanterade positiva Laplacian-v\u00e4rden till skillnad fr\u00e5n standardutj\u00e4mningsk\u00e4rnor som kunde ge negativa v\u00e4rden vid korta avst\u00e5nd, vilket orsakade \u00f6kningar i partiklarnas relativa hastighet (M\u00fcller, Charypar och Gross 2003).

4.1.3 Kollisionshantering

Kollisionshanteringen kontrollerade vid varje tidssteg om en partikel var utanför behållarens gränser och korrigerade partikelns position och hastighet enligt pseudokoden nedan.

```
BoundaryCollisions(){
```

```
För varje axel a = (x,y,z)
```

```
    relativHastighet = hastighet[i].a – behållarenHastighet[i].a
```

```
    if position[i].a < minGräns.a
```

```
        position[i].a = minGräns.a + offset
```

```
        hastighet[i].a = relativHastighet * dämpning + behållarenHastighet.a
```

```
    else if position[i].a > maxGräns.a
```

```
        position[i].a = maxGräns.a – offset
```

```
        hastighet[i].a = -relativHastighet * dämpning + behållarenHastighet.a
```

```
}
```

Behållarens hastighet subtraherades från partikelns hastighet innan spegling och adderades därefter tillbaka. Detta gav korrekt kollisionsrespons även när behållaren förflyttades under simuleringens gång. En *boundaryOffset* på 0,001 säkerställde att partiklar inte fastnade inuti väggarna.

4.2 GPU-implementeringen

4.2.1 Arkitektur och buffertar

GPU-implementeringens compute shader innehöll tio kernels som anropades i en fast ordning vid varje tidssteg: K1 (ClearForces), K2 (ComputeKeys), K3 (ClearCellStart), K4 (BuildCellStart), K5 (ComputeDensity), K6 (ComputePressure), K7 (ComputePressureForce), K8 (ComputeViscosity), K9 (ComputeForces) och K10 (BoundaryCollisions).

GPU-implementeringens utveckling påbörjades efter att den naiva versionen av CPU-implementeringen hade implementerats. GPU-versionen implementerades i ett separat skript med en tillhörande compute shader. De CPU-baserade arrayerna ersattes med ComputeBuffers för kommunikation mellan skriptet och compute shadern. Samma SPH-metoder från CPU-versionen implementerades som compute shader kernels K1-K10 i GPU-versionen.

En viktig förändring från CPU-versionen till GPU-versionen var hanteringen av kraftnollställning. I CPU-versionen nollställdes kraftarrayen i en separat loop i början av varje tidssteg. I GPU-versionen gjordes detta som K1, vilket kördes före övriga kernels. Om K1 inte hade körts separat, utan i stället varit en del av en annan kernel, hade krafterna från föregående tidssteg adderats till det aktuella tidssteget, vilket hade orsakat kraftökning och instabilitet i simuleringen.

4.2.2 Tidsstegshantering

Ett fast tidssteg, `fixedDeltaTime`, lades in i båda simuleringarna. Tjernell & Andersson (2023) noterade att SPH-simuleringar blev instabila vid stora tidssteg, något som uppstod när det inte fanns tillräckligt med grannpartiklar för uppskattning av densiteten. Med Unitys `Time.deltaTime` varierade tidssteget per bildruta, vilket orsakade att partiklarna rörde sig för långt per tidssteg. Detta resulterade i explosiva tryckkrafter samt generell instabilitet i simuleringen. Det fasta tidssteget sattes till 0,005, vilket vid visuell inspektion passade bäst för båda simuleringarna.

Det fasta tidssteget på 0,005 sekunder innebär att simuleringen behöver köras ungefär 3 gånger per bildruta vid 60 bilder per sekund (16,6 ms / 5 ms \approx 3,3 steg). Realtidsgränsen per tidssteg blir därmed inte 16,6 ms utan ungefär 5 ms.

4.2.3 Uniform Grid med spatial hashing

För att undvika den naiva $O(n^2)$ tidskomplexiteten implementerades Uniform Grid med spatial hashing enligt Green (2010) och Lague (2023), med en modifikation från Lague (2023) som tog bort behovet av att känna till rutnätets dimensioner i förväg. Implementeringen av spatial hashing bestod av fyra steg:

1. Beräkning av cellkoordinater för varje partikel
2. Omvandling av koordinater till hashvärden genom printalsmultiplikation
3. Mappning till ett nytt cellnyckelindex genom modulo operation
4. Sortering av partikelindexen baserat på cellnycklar

I GPU-versionen delades byggandet av rutnätet upp i tre separata kernels, `K2`, `K3` och `K4`, till skillnad från CPU-versionen där allt hanterades i en enda funktion. Anledningen var att GPU-kernels inte kunde synkronisera trådar mellan olika dispatch-anrop, vilket innebar att alla trådar i ett kernel-anrop behövde ha avslutat sin beräkning innan nästa kernel läste från samma buffert. Om dessa steg kombinerades i en enda kernel hade det orsakat race conditions där trådar läste data som ännu inte hade skrivits av andra trådar.

I CPU-versionen hanterades sorteringen med `Array.Sort` direkt på CPU:n. I GPU-versionen sorterades data på CPU:n, vilket krävde att cellnycklar överfördes från GPU:n till CPU:n och sedan skickades tillbaka sorterade till GPU:n. Denna dataöverföring skapade en flaskhals i GPU-implementeringen eftersom dataöverföringen mellan GPU och CPU var kostsam och långsam.

4.3 Prestandamätning

Mätrutinen mätte tiden per tidssteg med hjälp av *Stopwatch*. Mätningen startade när CPU:n körde `K1` och avslutades när *AsyncGPUReadback* bekräftade att GPU:n hade avslutat `K10`. Det som mättes var den totala beräkningstiden per tidssteg. Mätningen inkluderade GPU-beräkningstiden men exkluderade rendering och visualisering. Antalet mätpunkter sattes till 500 för att säkerställa ett tillräckligt stort underlag vid beräkning av stabila medelvärden. De första 50 mätpunkterna exkluderades för att undvika uppvärmningseffekter eftersom systemet ännu inte hade nått ett stabilt tillstånd. Medelvärde, minimum och maximum rapporterades för varje testat partikelantal.

En begränsning med den valda mätmetoden var att AsyncGPUReadback gav en viss flaskhals eftersom det krävde att GPU:n signalerade till CPU:n när all GPU-arbete hade slutförts. Denna flaskhals bedömdes vara försumbar jämfört med simuleringens beräkningstid och påverkade inte mätningarnas jämförbarhet mellan CPU-versionen och GPU-versionen. Ytterligare en begränsning var att dessa mätningar utfördes på en bärbar dator med integrerad GPU, Intel UHD Graphics med 128 MB dedikerat minne. Det innebar att den integrerade GPU:n hade betydligt lägre prestanda än en dedikerad GPU. Resultaten skulle av denna anledning se annorlunda ut på en annan dator. Prestandajämförelsen mellan CPU-versionen och GPU-versionen borde därför läsas med denna hårdvarubegränsning i åtanke. Mätresultaten representerade därmed den relativa prestandaskillnaden mellan CPU- och GPU-implementeringarna.

5 Resultat

Detta kapitel visar exekveringstiderna för de två SPH-implementeringarna. Avsnitt 5.1 presenterar CPU-implementeringens resultat, avsnitt 5.2 presenterar GPU-implementeringens resultat, avsnitt 5.3 jämför de båda implementeringarna och avsnitt 5.4 analyserar resultaten. Resultaten visas både i tabellform och som grafer. Tabellerna visar medelvärde, minimi- och maximivärde per antal partiklar, vilket möjliggör validering och numerisk jämförelse med andra studier. Graferna visualiserar skalningen och prestandaskillnaderna för att underlätta förståelsen av dessa.

5.1 CPU-implementering

I detta avsnitt presenteras exekveringstiderna för CPU-implementeringen vid de sex testade partikelantalen. Tabell 1 visar medelvärde, minimivärde och maximivärde i millisekunder och jämförelsegränsen är satt till 16,6 ms.

Partiklar	Medel (ms)	Min (ms)	Max(ms)
1 000	7,6851	5,9491	10,7687
5 000	37,0986	33,3610	47,1169
10 000	82,8402	73,5717	110,7141
25 000	253,2867	194,5999	325,8268
50 000	549,5762	440,4526	620,0925
75 000	863,6741	701,3946	1109,9651

Tabell 1: CPU-implementeringens exekveringstider

Vid 1 000 partiklar mättes en medeltid på 7,69 ms, vilket låg under jämförelsegränsen. Vid 5 000 partiklar överstegs samma gräns med en medeltid på 37,10 ms. Vid 10 000 partiklar mättes en medeltid på 82,84 ms och vid 25 000 partiklar 253,29 ms. Vid 50 000 partiklar mättes en medeltid på 549,58 ms och vid 75 000 partiklar 863,67 ms.

5.2 GPU-implementering

I detta avsnitt presenteras exekveringstiderna för GPU-implementeringen vid de sex testade partikelantalen. Tabell 2 visar medelvärde, minimivärde och maximivärde i millisekunder och jämförelsegränsen är satt till 16,6 ms.

Partiklar	Medel (ms)	Min (ms)	Max(ms)
1 000	9,6392	8,0606	18,2378
5 000	9,7489	8,0582	18,3319
10 000	12,442	8,5909	23,1283
25 000	22,4430	14,1763	30,6108
50 000	37,9219	22,3246	50,1635
75 000	52,1872	48,3572	63,0965

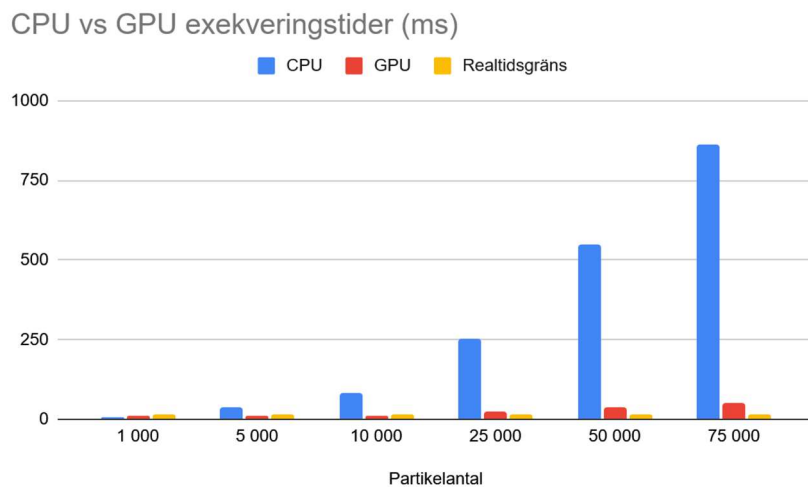
Tabell 2: GPU-implementeringens exekveringstider

Vid 1 000 och 5 000 partiklar uppmättes medeltider på 9,64 ms respektive 9,75 ms. GPU-versionen höll sig under jämförelsegränsen upp till och med 10 000 partiklar med en medeltid på 12,44 ms. Som beskrevs i avsnitt 4.2.2 var den praktiska realtidsgränsen per tidssteg ungefär 5 ms vid 60 bilder per sekund, vilket varken CPU- eller GPU-versionen klarade vid detta partikelantal. Vid 25 000 partiklar uppmättes en medeltid på 22,44 ms, vilket översteg jämförelsegränsen med 5,84 ms. Vid 75 000 partiklar uppmättes en medeltid på 52,19 ms.

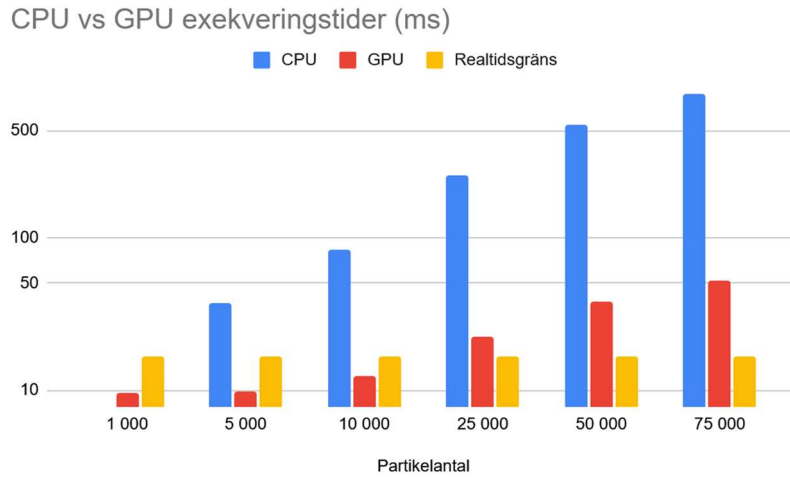
5.3 Jämförelse CPU och GPU

Det bör noteras att partikelantalen på x-axeln inte var jämnt fördelade eftersom stegen var 1 000, 5 000, 10 000, 25 000, 50 000 och 75 000. Exekveringstiderna ökade därför inte linjärt med partikelantalet på grund av den ojämna samplingen. CPU-versionens exekveringstid ökade i enlighet med den förväntade $O(n \log n)$ tidskomplexiteten för Uniform Grid med spatial hashing, vilket innebar att beräkningstiden växte snabbare än linjärt med partikelantalen. GPU-versionen uppvisade en mindre ökning vid låga partikelantal jämfört med CPU-versionen, vilket indikerade GPU:ns fasta overhead. Vid höga partikelantal ökade GPU-versionens beräkningstid markant, till följd av den växande beräkningsbelastningen och CPU-sorteringens stigande kostnad.

Figur 8 visar exekveringstiderna för CPU-versionen och GPU-versionen vid samtliga partikelantal, med jämförelsegränsen markerad som den gula stapeln. Figur 9 presenterar samma stapeldiagram i logaritmisk skala för att tydliggöra skillnaderna vid låga partikelantal.

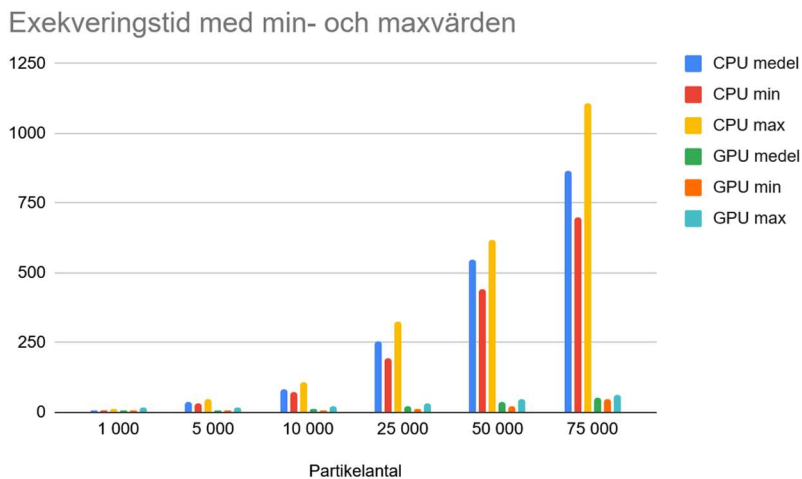


Figur 8: CPU vs GPU exekveringstider (ms)

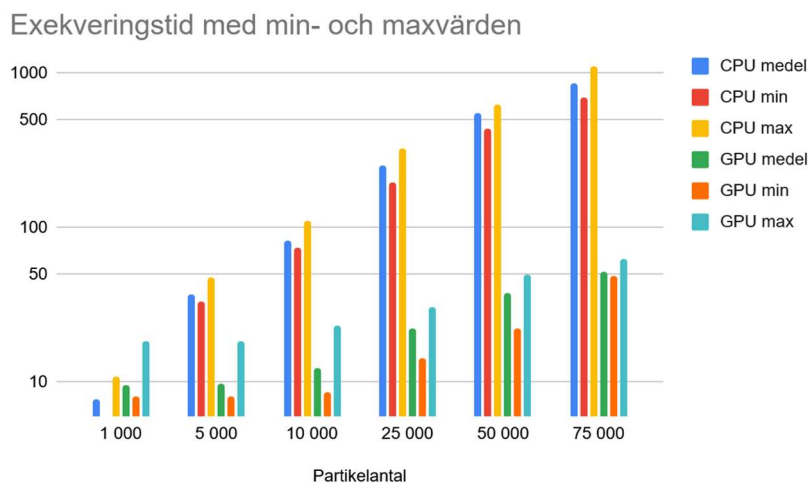


Figur 9: CPU vs GPU exekveringstider (ms) i logaritmisk skala på vertikala axeln för tydlighet

I Figur 10 och 11 nedan visas samma data som i Figur 8 och 9, men med min- och maxvärden inkluderade.

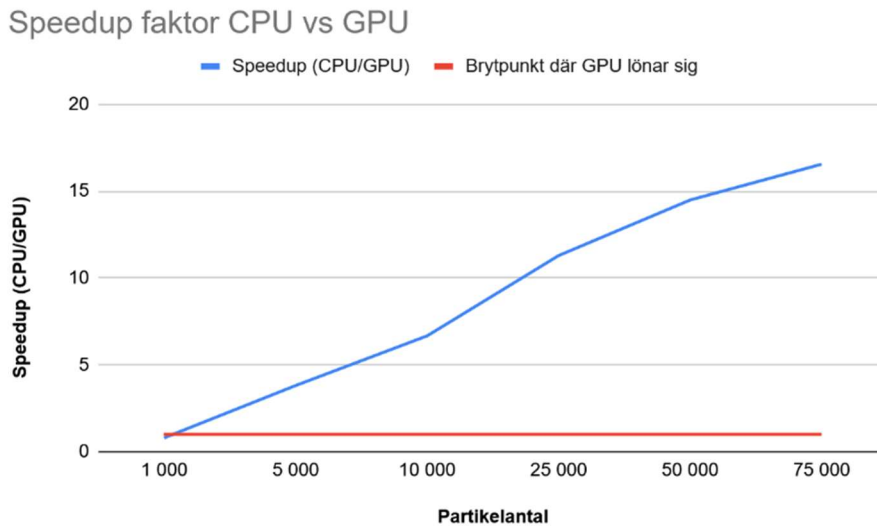


Figur 10: Exekveringstiderna med min- och maxvärden



Figur 11: Exekveringstiderna med min- och maxvärden i logaritmisk skala på vertikala axeln för tydlighet

I Figur 12 nedan visas speedup-faktorn (CPU-medeltid / GPU-medeltid) per partikelantal. Figuren illustrerar även hur GPU-fördelen ökar med problemstorleken.



Figur 12: Speedup (CPU-medeltid / GPU-medeltid) per partikelantal

Figur 8 och 9 visar prestandaskillnaden mellan CPU-versionen och GPU-versionen vid samtliga partikelantal. Den gula stapeln markerar jämförelsegränsen på 16,6 ms. Figur 10 och 11 visar samma data med min- och maxvärden inkluderade. Figur 12 visar speedup-faktorn (CPU-medeltid / GPU-medeltid) per partikelantal. Vid 1 000 partiklar uppmättes en speedup på 0,80, vid 5 000 partiklar 3,81, vid 10 000 partiklar 6,67, vid 25 000 partiklar 11,29, vid 50 000 partiklar 14,49 och vid 75 000 partiklar 16,55.

5.4 Analys av resultat

CPU-versionens exekveringstid ökar med den förväntade $O(n \log n)$ tidskomplexiteten för Uniform Grid med spatial hashing (Green 2010). När antalet partiklar ökade från 5 000 till 50 000 partiklar ökade även exekveringstiden med en faktor på 14,8. Detta tyder på att minnesrelaterade effekter uppstår vid högre partikelantal, troligen till följd av att CPU:n behöver hämta data som inte längre finns i cachén, vilket bidrar till den längre exekveringstiden. Detta stämmer med Waseem och Hong (2025), som påpekar att stora partikelbaserade simuleringar kräver noggranna optimeringar för att undvika minnesbegränsningar.

Det fasta tidssteget på 0,005 sekunder innebär ungefär 3 tidssteg per bildruta vid 60 bilder per sekund, vilket innebär att realtidsgränsen per tidssteg är ungefär 5 ms. På grund av detta uppfyller inte CPU-versionen eller GPU-versionen realtidskravet vid de flesta partikelantalen. Studiens realtidsgräns på 16,6 ms per tidssteg är ett förenklat jämförelsevärde och slutsatserna om vilket partikelantal uppfyller realtidskraven bör tolkas i förhållande till detta.

Den låga speedup-faktorn på 0,80 när GPU-versionen kördes med 1 000 partiklar kan förklaras av den fasta overheaden för kernel dispatch och av rutnätskonstruktionen (K2-K4). När antalet partiklar ökade användes GPU:ns parallella beräkningskapacitet mer effektivt, vilket troligen bidrog till den fortsatta ökningen av speedup-faktorn. GPU-versionen höll sig under jämförelsegränsen på 16,6 ms upp till 10 000 partiklar (12,44 ms), medan CPU-versionen passerade samma gräns vid 5 000 partiklar (33,1 ms). Den faktiska realtidsgränsen per tidssteg vid 60 bilder per sekund var ungefär 5 ms (se avsnitt 4.2.2), vilket innebär att ingen av versionerna uppfyllde realtidskraven vid dessa partikelantal. Speedup-faktorn ökade från 3,81 vid 5 000 partiklar till 16,55 vid 75 000 partiklar, vilket tyder på att GPU-parallellisering skalar väl med problemstorleken.

En viktig observation är att GPU-versionen håller sig under jämförelsegränsen på 16,6 ms upp till 10 000 partiklar, medan CPU-versionen överskrider samma gräns vid 5 000 partiklar. Resultaten antyder att GPU-versionen är mest användbar för spel vid höga antal partiklar, där realtidskraven är svåra att uppfylla med enbart CPU-versionen. Samtidigt ger GPU-versionen ingen prestandaförbättring vid lågt antal partiklar, mellan 1 000 och 5 000. Resultatet är relevant för spelutvecklare som överväger om en GPU-implementering är motiverad i enkla vattenscenarier. Resultaten tydde på att beslutet om en GPU-implementering kunde grundas på det partikelantal som spelscenariots vatten krävde.

Den stora skillnaden mellan denna studies GPU-resultat (52,19 ms vid 75 000) och GPU-resultaten från Hrytsyshyn m.fl. (2023) kan bero på flera faktorer: hårdvaruskilnaden (integrerad kontra dedikerad GPU), CPU-sorteringens dataöverföringskostnad och eventuella skillnader i grannsökningsoptimering. Trots skillnaderna är prestandaförbättringen jämförbar i båda studierna, vilket stärker validiteten hos resultaten från Hrytsyshyn m.fl. (2023). Eftersom Hrytsyshyn m.fl. (2023) inte dokumenterar sin hårdvara eller sina optimeringsval går det inte att avgöra hur mycket av skillnaden som beror på hårdvara respektive implementeringsval. Denna osäkerhet visar vikten av reproducerbarhet inom forskning och motiverar denna studies dokumentationsval.

6 Sammanfattning och diskussion

Detta kapitel sammanfattar studiens resultat och diskuterar dessa i relation till tidigare forskning. Avsnitt 6.1 ger en sammanfattning av genomförandet och resultaten. Avsnitt 6.2 diskuterar resultatens trovärdighet i relation till Hrytsyshyn m.fl. (2023) samt generaliserbarheten av resultaten. Avsnitt 6.3 behandlar samhälleliga och etiska aspekter. Avsnitt 6.4 ger förslag på framtida arbete.

6.1 Sammanfattning

Denna studie undersöker prestandaskillnaden mellan en CPU-baserad och en GPU-baserad implementering av Smoothed Particle Hydrodynamics (SPH) i spelmotorn Unity. Båda versionerna hade Uniform Grid med spatial hashing som grannsökningsoptimering. Studiens syfte är att validera resultaten från Hrytsyshyn m.fl. (2023) om GPU:ns prestandafördelar, eftersom deras studie saknar tillräcklig dokumentation för att möjliggöra reproducerbarhet.

Implementeringen baserades på Müller, Charypar och Gross (2003). SPH-algoritmen implementerades i två versioner: en CPU-version i C# och en GPU-version med compute shaders i HLSL. Båda versionerna använde Ekvation 2 för densitetsberäkningar, Ekvation 3 för tryckkraftsberäkningar och Ekvation 4 för viskositetsberäkningar, samt den symmetriska WCSPH-tryckkraftsformeln för att undvika partikelkluster.

Kvasi-experimentet kördes på en bärbar dator med en integrerad GPU (Intel UHD Graphics). Kvasi-experimentet mätte genomsnittlig exekveringstid i millisekunder per tidssteg för 1 000, 5 000, 10 000, 25 000, 50 000 och 75 000 partiklar, där jämförelsegränsen fastställdes till 16,6 ms.

CPU-versionen låg under jämförelsegränsen enbart vid 1 000 partiklar, med en medeltid på 7,69 ms, medan GPU-versionen låg under samma gräns upp till 10 000 partiklar, med en medeltid på 12,44 ms vid 10 000. Vid 75 000 partiklar var CPU-tiden 863,67 ms medan GPU-tiden var 52,19 ms, vilket resulterade i en speedup-faktor på 16,55. Resultaten bekräftade att GPU-versionen gav en stor prestandaförbättring jämfört med CPU-versionen, vilket överensstämmer med Hrytsyshyn m.fl. (2023). Prestandan begränsades av att kvasi-experimentet utfördes på en dator med en integrerad GPU, men den relativa prestandaförbättringen stöder studiens huvudsakliga slutsats om GPU-accelerationens prestandafördelar vid SPH-simuleringar.

6.2 Diskussion

6.2.1 Resultatets trovärdighet och relation till tidigare forskning

Studios resultat visade en prestandavinst för GPU-implementeringen vid samtliga partikelantal förutom vid 1 000 partiklar, där CPU-versionen var snabbare (7,69 ms kontra 9,64 ms). Detta var förväntat och stämde överens med GPU-arkitekturens typiska beteende, eftersom overheaden för att anropa compute shadern vid låga partikelantal var relativt stor jämfört med beräkningsbelastningen. Uniform Grid med spatial hashing introducerade en fast uppstartskostnad per tidssteg, vilket troligen också bidrog till GPU-versionens högre exekveringstid vid 1 000 partiklar. En naiv $O(n^2)$ implementering hade möjligen visat fördelen med GPU:n vid detta partikelantal eftersom den fasta overheaden för bygandet av rutnätet

hade tagits bort.

Grannsokningsoptimeringen var nödvändig för skalbarhet vid höga partikelantal. Samtidigt påverkade den jämförelsen vid låga partikelantal på ett sätt som bör beaktas vid tolkning av speedup grafen.

Hrytsyshyn m.fl. (2023) visade att deras GPU-implementering uppdaterade 75 000 partiklar på cirka 10 ms jämfört med denna studies 52,19 ms. Skillnaden var betydande men förväntad givet denna studies hårdvara. Denna studie kördes på en integrerad GPU med delat minne, vilket utgjorde en stor prestandabegränsning jämfört med den dedikerade GPU:n som Hrytsyshyn m.fl. (2023) sannolikt använde. Trots detta var prestandaökningen densamma, vilket stöder validiteten i Hrytsyshyn m.fl. (2023) resultat.

Ytterligare en faktor som troligen påverkade skillnaden i mätvärden var att GPU-versionens sortering utfördes på CPU:n, vilket innebar att dataöverföring skedde från GPU:n till CPU:n och tillbaka. Waseem och Hong (2025) lyfter fram att stora partikelbaserade simuleringar kräver noggranna optimeringar för att uppnå realtidsprestanda. En GPU-baserad sorteringsalgoritm, exempelvis Bitonic Sort eller radix sort, skulle sannolikt ha förbättrat GPU-versionens prestanda och minskat skillnaden till Hrytsyshyn m.fl. (2023) resultat. Det borde noteras att beslutet att behålla CPU-sorteringen motiverades av att båda implementeringarna skulle vara algoritmiskt identiska, men detta val skapade ett problem. CPU-versionen gynnades av att sorteringen skedde på CPU:n, medan GPU-versionen påverkades negativt av dataöverföringen. En rent GPU-baserad implementering utan detta krav hade sannolikt presterat mycket bättre, vilket innebar att denna studies GPU-resultat inte återspeglar GPU:ns fulla potential för SPH i Unity.

En annan faktor som påverkade speedup-faktorn var att CPU-versionen implementerades utan Unitys Burst Compiler, Jobs System eller DOTS. Dessa verktyg kunde ge stora prestandaförbättringar för SPH-simuleringen. CPU-versionen utnyttjade därmed inte CPU:ns fulla potential, vilket sannolikt bidrog till den uppmätta speedup-faktorn. En optimerad CPU-version skulle sannolikt ha minskat speedup-faktorn, vilket bör beaktas vid tolkning av resultaten.

Det fasta tidssteget på 0,005 sekunder innebär att simuleringen inte körs i realtid vid höga partikelantal. Tjernell och Andersson (2023) påpekar att SPH-simuleringar blir instabila när tidssteget är stort. För en speltillämpning krävs ett adaptivt tidssteg eller en substegs metod för att bibehålla stabilitet utan att realtidsprestanda försämras.

6.2.2 Generaliserbarhet

Resultaten bör tolkas med varsamhet vad gäller generalisering till andra datorer med olika hårdvara. Kvasi-experimentet kördes på en dator med integrerad GPU, vilket innebar att mätvärdena inte var direkt generaliserbara på datorer med dedikerad GPU. Det hade dock varit ett misstag att avfärda resultaten som inte generaliserbara i sin helhet. Den prestandaskillnad som innebär att GPU-versionen skalar bättre än CPU-versionen med ökande partikelantal var ett resultat som stöddes både av denna studie och av Hrytsyshyn m.fl. (2023). Ramakrishnan och McGraw (2023) samt Green (2010) lyfter också fram GPU-parallellisering som en nödvändighet för att SPH-simuleringen skulle uppnå realtidsprestanda. Att tre oberoende studier visade att GPU:n presterade bättre än CPU:n för SPH stärkte slutsatsen, även om den exakta prestandavinsten skiljde sig åt beroende på hårdvara.

6.3 Samhälleliga och etiska aspekter

Simulering av vatten är en viktig del av modern spelutveckling och vattensimulering påverkar miljontals spelare i spel som *Sea of Thieves* (2018), *Assassin's Creed IV: Black Flag* (2013) och *Still Wakes the Deep* (2024). En dokumenterad och reproducerbar jämförelse av en CPU-version och en GPU-version av SPH ger spelutvecklare praktisk kunskap när de överväger att använda SPH för simulering av vatten. Resultaten presenteras i Unity, en vanlig plattform för spelutveckling, vilket ökar tillämpligheten för många utvecklare.

SPH används även inom medicinsk visualisering och ingenjörssimulering. Metoder för att förbättra SPH-prestandan på konsumenthårdvara, inklusive integrerade GPU:er som finns i bärbara datorer, har därmed relevans även utanför spelindustrin. Studien fokuserade på att kunna återupprepas genom dokumentation av utjämningskärnor, hårdvara och optimeringsval. Detta bidrar till den vetenskapliga kunskapen på ett sätt som Hrytsyshyn m.fl. (2023) inte gör.

Utöver den samhälleliga nyttan finns även etiska aspekter att beakta. Studien använder inga människor eller djur, därför gäller inte de forskningsetiska principerna från Vetenskapsrådet (2002) om informationskrav, samtyckeskrav, konfidentialitetskrav och nyttjandekrav. I stället följer studien god forskningssed enligt Vetenskapsrådet (2024) genom tillförlitlighet i mätmetoden, ärlighet i redovisningen av resultat och ansvar för att dokumentera alla implementeringsval.

Tillgänglighet är också en viktig aspekt. Komplexa vattensimuleringar kräver en dedikerad GPU, vilket exkluderar spelare med svaga datorer från hela spelupplevelsen. Studien visar att en integrerad GPU kunde leverera realtidsprestanda med upp till 10 000 partiklar, vilket kunde räcka för vissa situationer i spel. Denna information är relevant för spelutvecklare som vill designa spel som fungerar på olika sorters datorer.

En aspekt som sällan diskuteras i spelutveckling, men som blir allt viktigare är energiförbrukningen vid GPU tunga simuleringar. En integrerad GPU förbrukar betydligt mindre energi än en dedikerad GPU, vilket innebär att de prestandavinster som GPU:n erbjuder kommer med en energikostnad. I takt med att molnbaserad spelstreaming växer, där simuleringar körs på kraftfulla servrar och strömmas till spelare, får denna energikostnad en större samhällelig relevans. Det uppstår därmed en spänning mellan strävandet efter visuell realism i spel och spelets miljöpåverkan, en avvägning som spelutvecklare och plattformägare behöver förhålla sig till.

Spelutvecklare möter ett svårt val. En mer beräkningsintensiv vattensimulering kräver mer energi och mer kraftfull hårdvara, vilket väcker frågan om vem som bär kostnaden. Kostnaden bärs antingen av spelaren som behöver uppgradera sin hårdvara eller av samhället genom ökad energiförbrukning i datacenter. Svaret beror på om simuleringen körs på spelarens dator eller via molnstreaming. Den lokala simuleringen lägger kostnaden på spelaren, medan den molnbaserade simuleringen lägger kostnaden på samhället. Båda fallen visar att bättre grafik aldrig är gratis. Denna insikt blir alltmer relevant i takt med att prestandakraven fortsätter att öka.

Samtidigt är det relevant att undersöka om 10 000 partiklar är tillräckligt för en bra simulering av vatten eller om 10 000 partiklar bara är en kompromiss som ger spelare med svaga datorer en sämre spelupplevelse. Frågan om var gränsen går mellan en acceptabel prestandaförsämring och en ojämlig spelupplevelse är både en teknisk och en etisk fråga som spelutvecklare behöver ta ställning till.

6.4 Framtida arbete

Som diskuteras i avsnitt 6.2 är CPU-sorteringen i GPU-versionen en flaskhals. Nästa steg är implementeringen av en GPU-baserad sorteringsalgoritm, exempelvis Bitonic Sort, direkt i compute shadern. Detta skulle sannolikt ge ett mer rättvisande mått på GPU-versionens fulla potential och skulle göra jämförelsen med Hrytsyshyn m.fl. (2023) enklare.

Utöver prestandaoptimering skulle det även ha varit värdefullt att komplettera studien med en visuell utvärdering, exempelvis via enkäter till spelare om hur realistiskt vattnet upplevdes. Som framgår av metoddiskussionen exkluderades detta medvetet, men en sådan studie skulle kunna ge svar på om den prestandagräns som identifierades, 10 000 partiklar för realtid på integrerad GPU, faktiskt var en visuellt acceptabel simulering för spel eller om prestandagränsen och den visuella gränsen skiljde sig åt.

Ett annat steg är att köra samma kvasi-experiment på en dator med dedikerad GPU. Detta kvasi-experimentet skulle ha förtydligat hur mycket av prestandaskillnaden som berodde på hårdvaran respektive på optimeringsval samt ha möjliggjort en mer pålitlig jämförelse med Hrytsyshyn m.fl. (2023).

Det skulle även vara värdefullt att undersöka adaptiva tidssteg för att uppnå realtidsstabilitet vid höga partikelantal samt att mäta hur renderingen av partiklarna påverkar den totala exekveringstiden i ett spelscenario, något som denna studie medvetet exkluderade.

På lång sikt är det relevant att sätta in SPH i en bredare kontext inom spelutveckling. Hybridmetoder som kombinerar SPH med Eulerian-baserade metoder kan användas för att utnyttja styrkorna hos båda metoderna. En framtida studie skulle kunna jämföra en GPU-baserad SPH-implementering med en hybridmetod som den Ramakrishnan och McGraw (2023) föreslog, med fokus på visuell kvalitet och prestanda. En sådan studie skulle också kunna besvara den fråga som lyfts i avsnitt 6.3 huruvida prestandagränsen på 10 000 partiklar är en visuellt acceptabel simulering eller om en hybridmetod skulle kunna uppnå likvärdig visuell kvalitet vid lägre beräkningskostnad.

I takt med att GPU:er och spelmotorer som Unity utvecklas mer, kan avancerade vattensimuleringar bli mer tillgängliga. När GPU:er blir starkare och spelmotorer får bättre stöd för GPU-baserade beräkningar blir körning av stora partikelsimuleringar i realtid enklare. Utvecklingen påverkar problemet med tillgänglighet som nämns i avsnitt 6.3. Om GPU:n i vanliga datorer blir starkare kan vattensimulering med SPH bli möjlig för spelare utan en dedikerad GPU.

Waseem och Hong (2025) pekar på att stora partikelbaserade simuleringar i Unity är ett aktivt forskningsområde och resultaten från denna studie bidrar med en dokumenterad baslinje mot vilken framtida implementering kan jämföras. I ett ännu större sammanhang följer denna utveckling ett mönster som setts i flera andra delar av datorgrafik. Avancerade tekniker som en gång krävde specialhårdvara, till exempel realtidsbelysning med ray tracing och fysikbaserad rendering, har blivit vanligare när hårdvaran har utvecklats. SPH för vattensimulering befinner sig i en tidig fas av denna utveckling.

Referenser

- Alklind Taylor, A. (2023). *Kvantitativ metod: Insamlingstekniker för kvantitativ data* [inspelad föreläsning]. *Kvantitativ metod: Insamlingstekniker för kvantitativ data - Play HS*
- Assassin's Creed IV: Black Flag* (2013). Ubisoft. Tillgänglig på Internet: <https://www.ubisoft.com/en-us/game/assassins-creed/IV-black-flag>
- AJTech (2023). *Fluid Simulations in Unity* [video]. Tillgänglig på Internet: <https://www.youtube.com/watch?v=9M72KrGhYuE&list=PL9FeLoYIHITxx3NmLpX8hR94pUuX5DtFE> [2026-03-05]
- Borg, E. & Westerlund, J. (2014). *Statistik för beteendevetare*. (3:e uppl.) Kina: Liber.
- Bridson, R. & Müller-Fischer, M. (2007). Fluid simulation: SIGGRAPH 2007 course notes. Video files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 courses (SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 1–81. <https://doi.org/10.1145/1281500.1281681>
- Christian, J. & Laursen, M. (2013). Improving the Realism of Real-Time Simulation of Fluids in Computer Games. <https://api.semanticscholar.org/CorpusID:61489698>
- Failes, Ian. (2013). *5 things you need to know about the tech of AC4*. <https://www.fxguide.com/featured/5-things-you-need-to-know-about-the-tech-of-assassins-creed-iv-black-flag/>
- Gingold, R. A. & Monaghan, J. J. (1977). Smoothed particle hydrodynamics: theory and application to non-spherical stars, *Monthly Notices of the Royal Astronomical Society*, Volume 181, Issue 3, December 1977, Pages 375–389, <https://doi.org/10.1093/mnras/181.3.375>
- Green, S. (2010). *Particle Simulation using CUDA*. Tillgänglig på Internet: <https://developer.download.nvidia.com/assets/cuda/files/particles.pdf>
- Hrytsyshyn, O., Venherskyi, P., Trushevskyi, V. & Terletsnyi, O. (2023). Smoothed Particle Hydrodynamics Implementation Using Compute Shaders. *2023 IEEE 13th International Conference on Electronics and Information Technologies (ELIT)*, 1–5. <https://doi.org/10.1109/ELIT61488.2023.10310973>
- Huang, H. & Yi, L. (2024). Journey into SPH Simulation: A Comprehensive Framework and Showcase. *ArXiv E-Prints*, arXiv:2403.11156. <https://doi.org/10.48550/arXiv.2403.11156>
- InteractiveComputerGraphics (2025). *Weakly compressible SPH (WCSPH)*. Tillgänglig på Internet: <https://interactivecomputergraphics.github.io/physics-simulation/examples/wcsph.html>
- Lague, S. (2023). *Coding Adventure: Simulating Fluids* [video]. Tillgänglig på Internet: <https://www.youtube.com/watch?v=rSKMYc1CQHE> [2026-03-05]
- Liu, M.B. & Liu, G.R. (2010). Smoothed particle hydrodynamics (SPH): An overview and recent developments, *Archives of Computational Methods in Engineering*, 17(1), pp. 25–76. doi:10.1007/s11831-010-9040-7.

- Lucy, L.B. (1977). A numerical approach to the testing of the fission hypothesis, *aj*, 82, pp. 1013–1024. Available at: <https://doi.org/10.1086/112164>
- Müller, M., Charypar, D. & Gross, M. (2003). Particle-based fluid simulation for interactive applications. *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 154–159.
- Natividad, S. (2022). *Games With The Best Water Physics* [Bild]. <https://gamerant.com/video-games-best-water-physics/>
- Nasss3x (u.å.) <https://wallpapercave.com/w/wp14267785>
- Ramakrishnan, V. & McGraw, T. (2023). *Water Animation Using Coupled SPH and Wave Equation*. Springer Science and Business Media Deutschland GmbH. doi:10.1007/978-3-031-47969-4_24.
- Sea of Thieves* (2018). Rare. Tillgänglig på Internet: <https://www.seaofthieves.com/>
- Still Wakes the Deep* (2024). The Chinese Room. Tillgänglig på Internet: https://store.steampowered.com/app/1622910/Still_Wakes_the_Deep/
- Tjernell, E. & Andersson, R. (2023). *Comparison between Smoothed-Particle Hydrodynamics and Position Based Dynamics for real-time water simulation*. Kandidatuppsats, Civilingenjör i datateknik. Kungliga Tekniska högskolan. <https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1795915&dswid=8241>
- Vetenskapsrådet (2002). *Forskningsetiska principer inom humanistisk-samhällsvetenskaplig forskning*. Stockholm: Elanders Gotab. *Vetenskapsrådet - Forskningsetiska principer*
- Vetenskapsrådet (2024). *God forskningssed 2024*. Tillgänglig på internet: <https://www.vr.se/analys/rapporter/vara-rapporter/2024-10-02-god-forskningssed-2024.html>
- Waseem, M. & Hong, M. (2025). Moving Towards Large-Scale Particle Based Fluid Simulation in Unity 3D, *APPLIED SCIENCES-BASEL*, 15(17), p. 9706. doi:10.3390/app15179706.
- West, M. (2008). *Practical Fluid Dynamics: Part 1 – GameDeveloper.com*. 26 Juni 2008. Practical Fluid Dynamics. Part 1. Tillgänglig på Internet: <https://www.gamedeveloper.com/programming/practical-fluid-dynamics-part-1> [Hämtad 2026-03-05]
- Wheater, Joe. (2024). *Making waves: developing realistic water mechanics for Still Wakes the Deep in UE5 – UnrealEngine.com*. 7 November 2024. Making waves: developing realistic water mechanics for *Still Wakes the Deep* in UE5. Tillgänglig på internet: <https://www.unrealengine.com/en-US/spotlights/making-waves-developing-realistic-water-mechanics-for-still-wakes-the-deep-in-ue5> [Hämtad 2026-03-05]
- Wikipedia (2026). *Smoothed-particle hydrodynamics*. Tillgänglig på internet: https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics