Examensarbete

HÖGSKOLAN I SKÖVDE

NAVIER-STOKES-BASERAD ELD I UNITY

Prestanda hos tvådimensionell eldsimulering i Unity

NAVIER-STOKES BASED FIRE IN UNITY

Performance of two dimensional fire simulation in Unity

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2024

Anton Andersson
Joel Carlsson

Handledare: Mikael Thieme
Examinator: Andreas Jonasson

Sammanfattning

I det här arbetet undersöks prestandapåverkan från en tvådimensionell Eulersk eldsimulering i screen-space i spelmotorn Unity. I simuleringen imiteras rörelser i den tredje dimensionen genom en uppskalning av tryckgradienten. Därefter utvärderas den prestandamässiga lämpligheten för metoden vid eventuell användning i datorspel. En artefakt utvecklades för att kunna utforma ett experiment där genomsnittlig beräkningstid för simulering av en bildruta mäts på GPU:n.

Studien visar att det finns en avsevärd negativ prestandapåverkan, men att denna påverkas till stor del av simuleringens upplösning samt antal iterationer i ett av simuleringsstegen, och slutsatsen dras att metoden i många fall kan vara lämplig för användning i datorspel beroende på dessa faktorer.

Vidare forskning föreslås där undersökningar görs på de visuella och estetiska implikationerna av metoden där fokus ligger på att redogöra för visuella artefakter beroende på kontexten för eldsimuleringen.

Nyckelord: Vätskesimulering, Navier-Stokes, Eld, Prestanda, Unity, Screen-Space

Innehållsförteckning

1 Introduktion 1
2 Bakgrund 2
2.1 Simulering av fluider 2
2.2 "Screen-space-animering av eld" 7
2.3 "GPU Gems" & Unity-implementation 8
2.4 Vätskesimulering i spel 9
3 Problemformulering 10
3.1 Frågeställning 11
3.2 Metod 11
3.3 Artefakt 12
3.4 Datainsamling 13
3.5 Metoddiskussion 14
4 Genomförande och analys 15
4.1 Artefakt 15
4.1.1 Unity-scen 15
4.1.2 MonoBehaviour 16
4.1.3 Texturer 18
4.1.4 Shaders 20
4.2 Experiment 28
4.2.1 Verktyg 28
4.2.2 Utförande 29
4.2.3 Framställning av grafer 29
4.3 Resultat 29
4.4 Analys 32
5 Sammanfattning och diskussion 37
5.1 Sammanfattning 37
5.2 Diskussion 37
5.3 Samhälleliga och etiska aspekter 38
5.4 Framtida arbete 39
Referenser 40

1 Introduktion

Navier-Stokes ekvationer ligger, som beskrivs av Harris(2007), till grund för många typer av vätskesimuleringar och så även för metoden som presenteras i detta arbete. En Eulersk tolkning av ekvationerna gör enligt Harris(2007) och Stam(2003) gällande att en fluid kan beskrivas med hjälp av en serie diskreta rutnät där varje cell representerar aktuellt tillstånd för fluidens olika egenskaper vid motsvarande position i simuleringen. Rutnäten kan manipuleras och förändringar över tid beräknas enligt Navier-Stokes.

Bakgrunden till detta arbete är en metod för simulering av eld som genom eulersk vätskesimulering baserad på Navier-Stokes ekvationer simulerar en realistisk eldeffekt på grafikkortet. Metoden demonstrerar flera visuella fördelar jämfört med traditionella metoder för eldeffekter som till exempel partikelsystem med hänsyn till att på ett trovärdigt sätt efterlikna de egenskaper som kan observeras hos verklig eld (Guay, Colin och Egli 2011).

Eftersom vätskesimulering enligt Stam(2003) ofta är förknippat med relativt dålig prestanda i realtidssammanhang formuleras frågeställningen i detta arbete gällande vilken typ av prestandapåverkan den metod som ligger till grund för arbetet har vid implementering i spelmotorn Unity och i förlängningen hur lämplig metoden kan tänkas vara för användning i datorspel.

I arbetet redovisas ett experiment för prestandamätning där en implementation som körs i huvudsak på grafikkortet konfigureras med olika upplösningar samt olika mängd iterationer i en prestandakritisk algoritm i simuleringen. Genomsnittstiden per bildruta för simuleringen över 10 000 uppdateringscykler för olika konfigurationer sett till upplösning och antal iterationer för Jacobi-metoden mäts med Unitys Recorder-API och medelvärden för dessa mätningar sammanställs för vidare analys med hjälp av diagram.

Resultaten visar att prestandapåverkan från eldsimuleringen är påtaglig men att prestandan effektivt kan styras genom att välja rätt konfiguration gällande upplösning och tidigare nämnda algoritmiterationer. En trend som påminner om ett linjärt förhållande mellan de oberoende variablerna och den uppmätta tiden kan tolkas ur diagrammen men vid närmare kontroll syns att relationen i själva verket inte är linjär.

2 Bakgrund

Det finns många olika sätt att efterlikna fenomenet eld i datorspel. Ett exempel på hur eld kan visualiseras i spelmotorn Unity är genom att spela upp en serie bilder av en komplett eld och sedan ständigt rotera denna så att den vänds mot kameran (Unity Technologies 2020). Ett annat exempel är hur partiklar kan kombineras med hjälp av ett partikelsystem för att skapa en dynamisk eldeffekt som på ett mer trovärdigt sätt smälter in i 3D-världen (Unity Technologies 2020).

Figur 1: Creating Fire with Particle Systems - Unity Technologies (2020)

2.1 Simulering av fluider

En helt annan metod för att simulera eld eller andra fluider i en datorspelsmiljö är genom så kallad vätskesimulering. På senare tid har fysikbaserade simuleringssystem för fluider som Unreal Engines Niagara lanserats vilket bland annat kan användas för simulering av eld och andra fluider (Epic Games u.å.). Niagara är bundet till spelmotorn Unreal Engine och går således inte att använda i andra motorer. Kellomäki (2017) undersöker flera olika metoder för att simulera fluider och konstaterade att endast ett begränsat antal 3D spel har implementerat detta. Kellomäki (2017) anser att de största hindren är prestanda, användarvänlighet och avsaknad av interaktion mellan fluider och andra objekt. Miao, Long, Miao, Qin (2023) undersöker utvecklingen inom området vätskesimulering och konstaterar hur kraven som sätts på både den visuella kvalitén samt kraven på interaktioner mellan fluiden och andra krafter ökat avsevärt. Miao (2023) nämner också hur vätskesimulering är ett av de mest utmanande forskningsområdena på grund av dess teoretiska komplexitet samt dess ofta tunga beräkningar.

Vätskesimulering lämpar sig väl till de fall där höga krav ställs på simuleringens fysikaliska trovärdighet (Stam 1999). Det vill säga att denna metod ofta ger mer trovärdiga resultat än tidigare nämnda lösningar där förrenderade bilder används. Green och Horvath (2012) nämner hur fördelar med simulerad eld bland annat är att den inte upprepar sig men de nämner också att en nackdel är den simulerade eldens beräkningstid.

Ett sätt att simulera eld, men även andra fluider, är genom algoritmer som bygger på Navier-Stokes ekvationer. Navier-Stokes ekvationer är som beskrivs av Stam (2003) en precis matematisk modell för de flesta vätskeflöden som förekommer naturligt. Stam (2003) nämner också hur de numeriska algoritmer som existerar för att lösa ekvationerna ofta är beräkningstunga då det i många fall ställs höga krav på simuleringens precision. Det nämns dock även hur det i datorgrafiksammanhang snarare ställs krav på visuell trovärdighet och snabb beräkningstid för vilket det i artikeln också presenteras en lösning.

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho} \nabla p + v\nabla^2 u + F$$

Ekvation 1: Icke-komprimerbara Newtonska fluider (Harris 2007)

$$\nabla \cdot u = 0$$

Ekvation 2: Preservation av massa (Harris 2007)

Detta arbete grundar sig i främst två av Navier-Stokes ekvationer. Ekvation 1 och ekvation 2 visar dessa. Harris (2007) tar upp hur dessa två Navier-Stokes ekvationer visar hur fluiden kan ändras över tid. I dessa två ekvationer är u är den konstanta fluiddensiteten, v är den kinematiska viskositeten och $F = (f_x, f_y)$ representerar externa krafter som påverkar fluiden. Som Harris (2007) även tar upp är det viktigt att notera att ekvation 1 faktiskt är två ekvationer då u är en vektorkvantitet, både ekvation 3 och 4 visar dessa.

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho} \nabla p + v\nabla^2 u + f_x$$

Ekvation 3: Icke-komprimerbara Newtonska Fluider (f_x) (Harris 2007)

$$\frac{\partial v}{\partial t} = -(u \cdot \nabla)v - \frac{1}{\rho} \nabla p + v\nabla^2 v + f_y$$

Ekvation 4: Icke-komprimerbara Newtonska Fluider (f_y) (Harris 2007)

För att förstå dessa ekvationer bättre kan man bryta ner dem i olika delar. Harris (2007) förklarar att de fyra olika termerna på högra sidan av ekvation 1 är accelerationer. Den första av dessa termen representerar självadvektionen och för att förklara detta ges exemplet hur velociteten av en fluid får fluiden att transportera objekt, densiteter och andra kvantiteter. Om man tänker sig att man häller till exempel färg i en fluid som rör sig blir färgen förflyttad, eller advekterad, längs fluidens velocitetsfält. Precis på samma sätt flyttar även fluiden sig själv precis som den hade flyttat färgen. Det är detta den första termen representerar. Termen kallas därför advektionstermen.

Den andra termen representerar enligt Harris (2007) accelerationen som sker när olika molekyler i fluiden interagerar med varandra. En kraft som appliceras på en fluid påverkar inte hela fluiden direkt, utan molekyler närmast kraften puttar på varandra och tryck byggs upp. Eftersom tryck är en kraft per enhetsområde byggs det naturligt upp en acceleration. Därför kallas den andra termen även för tryck-termen och representerar acceleration.

Den tredje termen i ekvation 1 representerar enligt Harris (2007) den faktor som gör att olika fluider är "tjockare" än andra. Till exempel att sirap rör sig långsamt medan vatten rör sig snabbare. Därför säger man att tjocka fluider har hög viskositet. Viskositet är ett mått på fluidens resistens mot flöde. Detta resulterar i en diffusion av moment som även kallas diffusionstermen.

Den fjärde och sista termen i ekvation 1 representerar enligt Harris(2007) acceleration som följd av externa krafter som påverkar fluiden. Dessa kan både vara lokala krafter eller kroppskrafter. Lokala krafter appliceras på ett specifikt område av fluiden, medan kroppskrafter appliceras jämnt på hela fluiden.

Operator	Definition	Ändlig differensform
Gradient	$\nabla p = \left[\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}\right]$	$\frac{P_{i+1,j} - P_{i-1,j}}{2\delta x}, \frac{P_{i,j+1} - P_{i,j-1}}{2\delta y}$
Divergens	$\nabla \cdot u = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$	$\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\delta x}$
Laplace	$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$	$\frac{P_{i+1,j} - 2P_{i,j} + P_{i-1,j}}{(\delta x)^2} + \frac{P_{i,j+1} - 2P_{i,j} + P_{i,j-1}}{(\delta y)^2}$

Tabell 1: Vektoranalys som används inom fluid simulering av Harris (2007)

Förutom Navier-stokes används enligt Harris(2007) även vektoranalys i lösningen vilket framgår i tabell 1. Som förklaras i Harris (2007) är gradienten av ett skalärfält en vektor som är en partiell derivata av skalarfältet. Divergens avser nettoflödet ut ur ett givet område. I Navier-Stokes ekvationer applicerar detta på hastigheten av flödet, och visar en skillnad i hastigheten på ett område omkring en ruta. Ekvation 2 som nämndes tidigare garanterar att vätskan är icke-komprimerbar genom att se till att vätskan alltid har noll divergens. Skalärprodukten i divergensoperationen resulterar i en summa av partiella derivator, vilket är detsamma som gradientoperationen. Detta innebär att divergensoperatorn endast kan appliceras till vektorfält. Det går här att dra slutsatsen som Harris (2007) tar upp att gradienten av ett skalärfält är ett vektorfält, och att divergensen av ett vektorfält är ett skalärfält. Om man applicerar divergensoperatorn på resultatet av gradientoperatorn, blir resultatet en Laplaceoperation. Alltså, om rutnätet följer förhållande ∂x = y kan man simplifiera Laplaceoperationen till ekvation 5 nedan (Harris 2007). Ekvationer av formen ∇^2 x= b kallas även Poissonekvationer och kommer hädanefter att refereras till som sådana.

$$\nabla^2 p = \frac{P_{i+1,j} + P_{i-1,j} + P_{i,j+1} + P_{i,j-1} - 4P_{i,j}}{(\delta x)^2}$$

Ekvation 5: Simplifierad Laplaceoperator om det antas att rutnätets celler är kvadrater (Harris 2007)

För att sedan kunna lösa Navier-Stokes ekvationer behöver de ändras till en form som lämpar sig bättre för numeriska lösningar. Vektoranalys visar hur en vektor v kan bli uppdelad i vektorkomponenter vars summa blir vektorn v. Till exempel, som Harris (2007) beskriver, kan man ta en vektor $v = (x, y)$ och skriva om den som v= $x\hat{\imath} + y\hat{\jmath}$ där î och ĵ är enhetsvektorerna för vardera axel. På samma sätt kan man dekomponera ett vektorfält till en summa av andra vektorfält. Helmholtz-Hodge-dekompositionssatsen i ekvation 6 visar hur ett vektorfält kan bli uppdelat till summan av två andra vektorfält, ett divergensfritt vektorfält, och en gradient av ett skalärfält.

$$w = u + \nabla p$$

Ekvation 6: Helmholtz-Hodge-dekompositionssats, där *u* har noll divergens (Harris 2007)

Med hjälp av dessa ekvationer kan man dra flera olika slutsatser relaterade till lösning av Navier-stokes ekvationer. Som Harris (2007) tar upp involverar Navier-Stokes ekvationerna

tre olika uträkningar för att uppdatera velociteten, advektionen, diffusionen och kraftappliceringen. Resultatet är ett nytt velocitetsfält, w, som inte är divergensfritt. Kontinuitetsekvationerna kräver dock att det är noll divergens. Därför kan man använda Helmholtz-Hodges dekompositionssats, vilken säger att divergensen av ett velocitetsfält kan korrigeras genom att subtrahera gradienten av tryckfältet från velocitetsfältet.

$$u = w - \nabla p$$

Ekvation 7: Korrigerad divergens av velocitetsfält genom subtraktion av gradienten av tryckfältet (Harris 2007)

Något annat som i Harris(2007) görs tydligt är en metod för att räkna ut tryckfältet. Genom att applicera divergensoperatorn till båda sidorna av Helmholtz-Hodge-satsen får vi resultatet i ekvation 8, vilket kan förenklas till ekvation 9, eftersom att ekvation 2 visar att $\nabla \cdot u = 0$. Resultatet blir Poissonekvationen för tryck i en fluid. Detta innebär att, givet ett divergent hastighetsfält w, kan man räkna ut p med hjälp av ekvation 9 och sedan använda w och p för att räkna ut det nya divergensfria fältet u via ekvation 7.

$$\nabla \cdot w = \nabla \cdot (u + \nabla p) = \nabla \cdot u + \nabla^2 p$$

Ekvation 8: Applikation av divergensoperatorn till båda sidorna av ekvation 6 (Harris 2007)

$$\nabla^2 p = \nabla \cdot w$$

Ekvation 9: Simplifierad version av ekvation 8 (Harris 2007)

Därefter används Helmholtz-Hodge-satsen för att definiera en projektionsoperator $\mathbb{P}$ som projicerar ett vektorfält w till dess divergensfria komponent u. Om $\mathbb{P}$ appliceras på ekvation 6 resulterar det i ekvation 10. Genom definitionen av $\mathbb{P}$, $\mathbb{P}w = \mathbb{P}u = u$, är alltså $\mathbb{P}(\nabla p) = 0$. Av detta följer att det går att förenkla Navier-Stokes ekvationer ytterligare. Först appliceras projektionsoperatorn till båda sidorna av ekvation 1 vilket resulterar i ekvation 11. Men eftersom u är divergensfritt är också derivatan i vänsterledet det. $\mathbb{P}(\partial u/\partial t) = \partial u/\partial t$. Tillsammans med $\mathbb{P}(\nabla p) = 0$ innebär detta att trycktermen tas bort och resultatet blir ekvation 12.

$$\mathbb{P}_w = \mathbb{P}u + \mathbb{P}(\nabla p)$$

Ekvation 10: Projektionsoperatorn P applicerad på ekvation 6 (Harris 2007)

$$\mathbb{P}\frac{\partial u}{\partial t} = \mathbb{P}\left[(-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + v\nabla^2 u + F\right]$$

Ekvation 11: Förenklad variant av ekvation 1 via applicering av projektionsoperatorn P (Harris 2007)

$$\frac{\partial u}{\partial t} = \mathbb{P}(-(u \cdot \nabla)u + v\nabla^2 u + F)$$

Ekvation 12: Förenklad version av ekvation 11 då u är divergensfri (Harris 2007)

I en faktisk implementation, som Harris (2007) förklarar, brukar olika komponenter inte beräknas samtidigt och adderas som i ekvation 12. Istället brukar beräkningarna delas upp i olika steg för varje komponent som tar ett fält som indata och beräknar ett nytt fält som utdata. Därför kan man definiera en uppdateringscykel av simuleringen genom operatorn $\mathbb{S}$, som är samma sak som en uträkning av ekvation 12 över ett tidssteg. Genom att definiera advektion som $\mathbb{A}$, diffusion som $\mathbb{D}$, kraftapplicering som $\mathbb{F}$, och projektion som $\mathbb{P}$ får vi ekvation 13. Värt att notera är att tidssteget för tydlighetens skull inte är med i ekvation 13, men i en faktiskt implementation krävs det i varje operation.

$$\mathbb{S}(u) = \mathbb{P} \circ \mathbb{F} \circ \mathbb{D} \circ \mathbb{A}(u)$$

Ekvation 13: Ett steg av simulerings algorithmen (Harris 2007)

För att kunna slutföra en korrekt simulering krävs även steg för att lösa advektion, diffusion samt en lösning för Poissonekvationerna. Som Harris (2007) tar upp innebär advektionsprocessen att en fluids hastighet transporterar sig själv samt andra egenskaper hos fluiden. För att räkna ut detta krävs en uppdatering på varje punkt i rutnätet. Processen för att räkna ut detta börjar med Eulers metod vilket visas i ekvation 14. Detta flyttar positionen r av varje punkt längs hastighetsfältet den distans den hinner på tiden t. Denna lösning har dock två problem. Det första är att metoden är instabil för stora lösningar och kan resultera i felaktiga resultat om $u(t)t$ är större än en cell i rutnätet. Det andra problemet är specifikt för GPU-implementationer. Eftersom simuleringen körs parallellt i fragment-delen i ett shader-program kan man inte ändra ett värde samtidigt som man läser samma värde från en textur. För att undvika detta använder Harris (2007) en implicit metod som presenteras av Stam (1999). Metoden innebär att istället för att räkna ut vart varje kvantitet rör sig vid varje tidssteg, kan man kopiera kvantiteten från en tidigare position till nuvarande position. För att uppdatera en kvantitet q, till exempel hastighet, densitet eller temperatur, används ekvation 15.

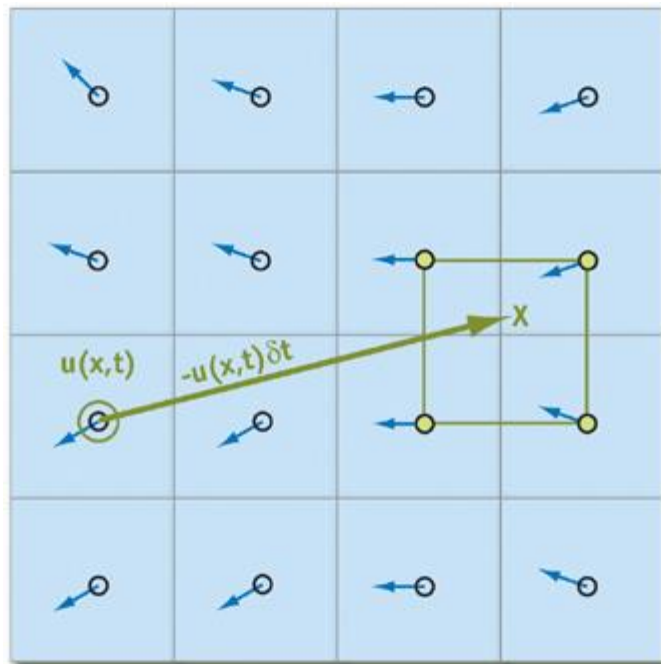
$$r(t + \delta t) = r(t) + u(t)\delta t$$

Ekvation 14: Eulers metod för integration av normala differentialekvationer (Harris 2007)

$$q(x, t + \delta t) = q(x - u(x,t)\delta t, t)$$

Ekvation 15: Advektion av en kvantitet q (Harris 2007)

Som Stam (1999) visar går detta lätt att implementera på GPU:n men det är också stabilt för arbiträra tidssteg och hastigheter. Detta visas i figur 2, vilket visar advektionberäkningen vid cellen markerad med en grön cirkel. Först spåras hastighetsfältet tillbaka i tiden vilket leder till det gröna x. De fyra rutorna närmast gröna x blir sedan bilinjärt interpolerade och resultatet skrivs tillbaka till den ursprungliga rutnätscellen.



Figur 2: Visualisering av advektion (Harris 2007)

För att räkna ut den partiella differentialekvationen för diffusion av viskositet används som Harris (2007) beskriver ekvation 16. För att undvika större värden som kan bli instabila används återigen Stam (1999) som gör en implicit formulering av ekvation 16, vilket resulterar i ekvation 17. I ekvation 17 är I en identitetsmatris. Denna formulering är som Harris (2007) beskriver stabil för arbiträra tidssteg och viskositeter. Detta resulterar i en Poissonekvation för hastighet.

$$\frac{\partial u}{\partial t} = v\nabla^2 u$$

Ekvation 16: Partiell differentialekvation för viskositetsdiffusion (Harris 2007)

$$(I - v\delta t\nabla^2)u(x, t + \delta t) = u(x, t)$$

Ekvation 17: Omformulering av ekvation 16 (Stam 1999)

Poisson-ekvationen är som Harris (2007) tar upp en matrisekvation i formen av Ax = b. Där x är en vektor av värden som v läser, b en vektor av konstanter och A är vår matris. För att lösa både Poissionekivationen för tryck och för viskositetsdiffusion används en iterativ lösning som börjar med en första gissning, $x^{(0)}$, där varje steg k itererar och producerar en förbättrad lösning $x^{(k)}$. Notationen indikerar iterationen. Den lättaste iterationsmetoden, vilken Harris (2007) presenterar, kallas Jacobi-metoden. En derivata av Jacobi-metoden presenteras av Harris (2007). Både ekvation 9 och ekvation 17 kan skrivas om i form av ekvation 18. Vad de olika variablerna betyder skiftar mellan båda lösningarna. I Poisson-tryck-ekvationen är $x = p$, b representerar $\nabla \cdot w$, a = -$(\delta x)^2$ och β = 4. För diffusionsekvationen för viskositet är både x och $b = u$, a = (δx)2/vδt, och β = 4 + α.

$$x_{i,j}^{(k+1)} = \frac{x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + ab_{i,j}}{\beta}$$

Ekvation 18: Formulering av ekvation 9 och 17 i en gemensam form för att tillåta iteration (Harris 2007)

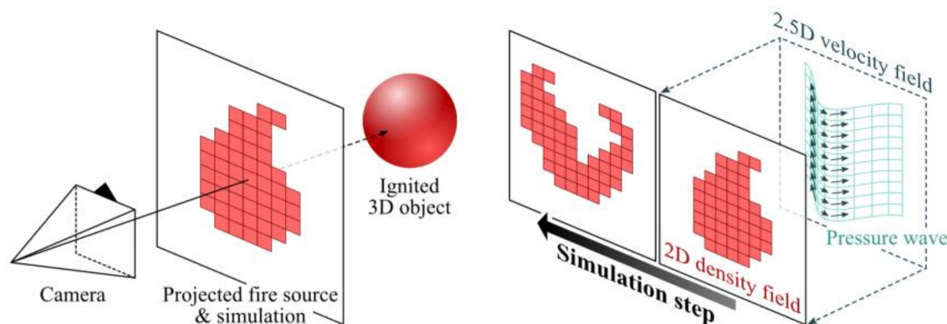
Genom att formulera ekvationerna på detta sätt tillåts användning av samma kod för båda ekvationerna. För att lösa ekvationerna som Harris (2007) skriver används alltså ekvation 18 på varje cell i rutnätet, och resultatet från den förra iterationen används som indata för nästa iteration. Som Harris (2007) förklarar konvergerar Jacobi-iterationerna snabbt. Därför måste många iterationer köras. Dock poängterar Harris (2007) också att detta är väldigt prestandaeffektivt på GPU:n.

Det bör poängteras att det finns olika metoder för att simulera fluider enligt Navier-Stokes principer. Detta arbete använder sig främst av metoder som grundar sig i en Eulersk tolkning av Navier-Stokes. De metoder som associeras med Euler beskriver i regel systemet för vätskesimuleringen genom en serie diskreta rutnät där det mest grundläggande för systemet är ett vektorfält som beskriver fluidens rörelse medan övriga rutnät kan bestå av antingen vektorer eller skalärer och beskriva andra egenskaper hos fluiden som till exempel densitet eller temperatur (Harris 2007).

2.2 “Screen-space-animering av eld”

I postern Screen Space Animation of Fire presenterar Guay, Colin och Egli (2011) en metod för att på ett effektivt sätt simulera eld baserat på Navier-Stokes ekvationer. Denna poster är den huvudsakliga inspirationen för vårt eget arbete och implementeringen av eldsimuleringen i vårt arbete konstrueras med målet att efterlikna den effekt som presenteras

i postern. Simuleringen sker i två dimensioner och har därför komplexiteten O($n^{(2)}$) vilket är en klar prestandamässig fördel gentemot en simulering i tre dimensioner där komplexiteten istället blir O($n^{(3)}$). En simulering i två dimensioner ger en mindre trovärdig simulering av eld då en dimension av rörelse saknas men i artikeln beskrivs en metod för hur rörelse i den saknade dimensionen kan imiteras med hjälp av tryckvågor eller vibrationer i ett av simuleringsstegen. Detta genom att till viss del göra ett undantag för villkoret för icke-komprimerbarhet som vanligen återfinns i Eulerska simuleringar av fluider. Detta undantag avser en uppskalning av tryckgradienten och därmed också de tryckvågor som uppstår vid tryckutjämnande beräkningar på simuleringen vilket ger effekten av att eldsflammorna i simuleringen 'bryts upp' och således ges en illusion av rörelse i djupled. Tryckutjämningen är det som sker vid subtrahering av tryckgradienten från velocitetsfältet.



Figur 3: "Left: An ignited object in the scene has its fire source value projected onto the simulation grid. Right: Pressure waves act on the shape of flames to create the illusion of 3D motion." Guay, Colin och Egli (2011)

Utöver detta beskrivs i artikeln även hur simuleringen kan köras i 'screen space' istället för att simuleras i 3D-scenen och rastreras tillsammans med resten av 3D-miljön. Detta genom att bränslekällor, objekt i scenen som skall ge upphov till eldsflammor, rastreras med ett värde för intensiteten på bränslet, vilket illustreras i figur 3. Med hjälp av detta värde kan eldsimuleringen köras i 'screen space' och simuleringen tar hjälp av intensitetsvärdet för att bestämma varifrån elden skall uppstå.

2.3 "GPU Gems" & Unity-implementation

Implementationen i detta arbete bygger i grund och botten på metoden för vätskesimulering med hjälp av GPU:n som beskrivs av Harris (2007). I artikeln tar Harris upp hur många av teknikerna han beskriver baserar sig i Stam (1999) men istället för att köra simuleringen på CPU:n som i Stam(1999) körs simuleringen här istället på GPU:n. Som Harris (2007) tar upp är grafikhårdvara väl anpassad till den typ av beräkningar som krävs för vätskesimulering. Det tas även upp hur grafikhårdvara är optimerad för att utföra beräkningar på pixlar, vilket är likt det rutnät av celler som simuleringen kräver. GPU:n uppnår hög prestanda genom att den kan processa flera vertiser och pixlar parallellt. Den är också optimerad för att genomföra flera texturläsningar per uppdateringscykel. Eftersom simuleringens rutnät lagras i texturer är denna hastighet och parallellisering precis vad som behövs i denna typ av simulering.

För experimentet i detta arbete krävdes en implementering av Harris(2007) metod som fungerar i spelmotorn Unity. För detta valdes en implementering som utvecklats av Hawkins (2017) specifikt för Unity baserat på Harris metod. Metoden är inte en exakt implementering av Harris metod, bland annat saknas beräkningen av diffusion av viskositet. Dock är Hawkins

metod mycket lik Harris metod då den är baserad på samma teoretiska bakgrund. Hawkins projekt omfattas av en MIT-licens och en bedömning gjordes gällande att delar av projektet kunde användas som bas för eldsimuleringen i detta arbete förutsatt att ett flertal modifikationer gjordes i enlighet med de fynd som presenteras i Guay, Colin och Egli (2011). Hawkins projekt kör en enkel vätskesimulering på GPU:n med hjälp av en serie shaders som utför de olika beräkningsstegen. Dessa shaders anropas från ett MonoBehaviour-script som bland annat ansvarar för simuleringens inställningar och tidssynkronisering för uppdateringscykeln.

2.4 Vätskesimulering i spel

I Guay, Colin och Egli(2011) påpekas hur deras metod inte ger en fysiskt korrekt eldsimulering men som nämns i Stam (2003) är det inte i första hand den fysiska precisionen som är viktig när det handlar om eldsimulering i datorgrafiksammanhang som bland annat datorspel utan istället är det den visuella trovärdigheten och simuleringens prestanda som spelar störst roll.

Kellomäki(2017) påpekar hur fysiskt korrekta vätskesimuleringar, år 2017, i regel fortfarande är alldeles för beräkningstunga för att kunna användas i datorspel och därför krävs ofta extrema förenklingar av de fysikaliska beräkningarna. Kellomäki(2017) presenterar också hur metoden för vätskesimulering i hans artikel, vid tiden för studien var snabbare än andra relativt komplexa metoder för vätskesimulering. Dock nämns i artikeln också hur man för att dölja att metoden inte är fysiskt korrekt har behövt använda sig av olika 'tricks' som shader-effekter eller minskad partikelmängd.

Miao et al(2023) beskriver hur utvecklingen inom fältet för vätskesimulering gått mycket fort. De nämner dock hur vätskesimulering fortfarande är mycket krävande i de fall då höga krav ställs på den fysiska korrektheten. De nämner också hur det är svårt att förenkla simuleringen av fluider då teorin bakom fluiders beteende bygger på de verkliga fysiklagarna bakom fenomenet. På grund av komplexiteten i både teorin bakom vätskesimulering samt komplexiteten i beräkningarna är vätskesimulering fortfarande ett av de mest utmanande forskningsområdena.

3 Problemformulering

Detta arbete fokuserar på eldsimulering baserat på Navier-Stokes ekvationer och specifikt på den metod som presenteras i Guay, Colin och Egli (2011) för att använda en tvådimensionell eulersk vätskesimulering som ett sätt att skapa en eldeffekt i screen-space i en i övrigt tredimensionell scen.

Fluider kan i regel representeras som ett vektorfält där momentanhastigheten för varje punkt i fältet representeras som en vektor. Det vill säga att varje punkt i den tänkta fluiden har en hastighet och riktning (Stam 2003). Navier-Stokes ekvationer beskriver hur ett sådant vektorfält kommer att förändras över tid. Stam beskriver också hur det för att visualisera förändringarna i vektorfältet behövs något medium som förflyttas baserat på vektorfältet. Detta skulle kunna vara enstaka partiklar enligt Lagrange-metoden men för att minska beräkningskostnaden är det lämpligt att visualisera fluider med hjälp av ett Eulerskt densitetsfält snarare än enstaka partiklar. Densitetsfältet består av ett skalärfält där densiteten för en tänkt fluid för varje punkt i fältet representeras med ett skalärt värde. Förändringar i densitetsfältet beräknas baserat på vektorfältets tillstånd. Genom att beräkna dessa förändringar stegvis kan man simulera fluidens rörelse och densitet. Genom att introducera förändringar i vektorfältet eller densitetsfältet kan man manipulera fluiden då simuleringen kommer att svara på förändringarna över tid.

Guay, Colin och Egli (2011) presenterar en metod för hur eld kan simuleras med hjälp av Navier-Stokes ekvationer. Elden i simuleringen representeras genom ett densitetsfält i screen-space där densiteten vid varje punkt motsvarar eldsflammornas intensitet. Densitetsfältets advektion baseras på ett tillhörande vektorfält. Ett eller flera objekt i scenen rastreras med ett värde som används för att introducera förändringar i densitetsfältet vilket ökar densiteten vid de punkter där elden skall ha sitt ursprung. Det som gör denna metod särskilt intressant är det faktum att trots att simuleringen sker i två dimensioner ger visualiseringen en illusion av att elden är tredimensionell. Detta genom att snabba vibrationer introduceras i vektorfältet. Dessa vibrationer modelleras som tryckvågor liknande de som återfinns i komprimerbara fluider genom att tryckgradienten i simuleringen skalas upp.

Mot bakgrund av att vätskesimulering som metod för att simulera eld enligt Kellomäki(2017) är förhållandevis ovanligt i datorspel kan det argumenteras för att metoden som beskrivs i Guay, Colin och Egli(2011) borde undersökas vidare för att utvärdera dess lämplighet för användning i just datorspel då den antyder att det går att simulera en visuellt trovärdig eld i realtid.

Det framgår i både Guay, Colin och Egli(2011) samt Harris(2007) att det bevisligen går att simulera fluider i realtid med förhållandevis bra prestanda redan vid tiden för dessa arbeten. Harris metod är redan implementerad i Hawkins(2017) Unity-projekt som dessutom går att modifiera för att upprepa de tekniker som presenteras i Guay, Colin och Egli(2011) då de båda implementationerna är av liknande karaktär och bygger på samma grundläggande metod för vätskesimulering. Detta motiverar en upprepning av Guay, Colin och Eglis metod för eldsimulering i Unity med Hawkins(2017) källkod som grund för att vidare kunna utvärdera metodens lämplighet för att simulera eld i datorspel. Det finns förstås en mängd faktorer som påverkar bedömningen av metodens lämplighet för användning i datorspel men som ett första steg föreslås i detta arbete en undersökning av prestandan, vilken också rimligtvis är en grundläggande förutsättning för att kunna använda eldsimuleringen i

praktiken. För att prestandamätningen skall ge en rättvis bild av metodens lämplighet läggs i detta arbete stor vikt vid att efterlikna den visuella effekten ur Guay, Colin och Egli(2011) så nära som möjligt.

3.1 Frågeställning

Frågeställningen som undersöks i detta arbete är: Vilken prestandapåverkan har en Eulersk simulering av en fluid baserad på metoden i Harris(2007) i Unity med syfte att efterlikna eldeffekten som beskrivs i Guay, Colin och Egli (2011)?

Värt att notera är att syftet med detta arbete inte är att jämföra prestandan på implementationen i detta arbete med den i Guay, Colin och Egli (2011) utan snarare att upprepa deras metod för att kunna dra slutsatser om metodens prestanda i den miljö som används för experimentet i detta arbete. Med andra ord är det experiment som presenteras i arbetet att betrakta som en upprepning av de modifieringar på den övergripande simuleringen som ger elden dess realistiska utseende vilket beskrivs i Guay, Colin och Egli (2011) men inte en upprepning av deras metod för prestandamätning eller val av exakta algoritmer för simuleringsstegen.

I det här arbetet avser användningen av begreppet prestanda den tid det tar för grafikkortet att köra ett beräkningssteg i eldsimuleringen. Denna tid mäts i nanosekunder med hjälp av Unitys Profiler-API där funktioner för att mäta beräkningstiden finns (Unity Technologies 2024).

Orsaken till att i detta arbete mäta prestandan avseende beräkningstiden på grafikkortet är för att en låg beräkningstid ger mer utrymme i tid för andra beräkningar som kan behövas i ett datorspel. Detta förhållande i beräkningstid framgår inte om enbart uppdateringsfrekvensen för simuleringen mäts. Om beräkningstiden för någon process i ett datorspel är för hög kan detta påverka uppdateringsfrekvensen vilken enligt Claypool och Claypool (2007) har en påverkan på spelarens upplevelse.

Guay, Colin och Egli (2011) noterar hur de för simuleringen använt sig av GPU-modellen GeForce 9800 GT. Detta grafikkort lanserades år 2008 (TechPowerUp u.å.). För vårt eget experiment används ett GeForce RTX 3070 som lanserades år 2020 (TechPowerUp u.å.). Mot bakgrund av detta samt att det i Guay, Colin och Egli (2011) och Harris(2007) redogörs för att deras metoder är prestandaeffektiva gjordes antagandet att även simuleringen i detta arbete kommer att prestera relativt väl. Detta är ett skäl till varför metoden upprepas på nyare hårdvara i detta arbete.

3.2 Metod

Metoden för detta arbete kretsar kring insamling av data genom att utföra ett experiment där mätinstrumentet i fråga är Unitys Recorder-API för mätning av exekveringstid och den data som skall samlas in den tid som krävs för att utföra ett helt steg, det vill säga hela beräkningen på grafikkortet för simuleringen som skall ske under en bildruta, i eldsimuleringen. Eftersom denna data kommer vara kvantifierbar kommer en analys av insamlad data genomföras med diagram som visuellt hjälpmedel.

I Gunadi och Yugopuspito (2018) undersöks skillnader mellan OpenGL och Vulkan i förhållande till en typ av vätskesimulering baserad på Navier-Stokes. I artikeln utförs ett experiment på ett område som angränsar det område som undersöks i detta arbete och därför kan man argumentera

för att metoden som presenteras i artikeln fungerar som ett bra exempel på hur denna typ av experiment kan utföras. Gunadi och Yugopuspito (2018) beskriver noggrant vilken hårdvara som används under experimentet vilket är något som är relevant även i detta arbete. Utöver detta tar de hjälp av olika sorters diagram för att tolka resultaten från deras datainsamling. Detta ger en god översikt över resultaten och underlättar sannolikt analysen.

Simuleringen som används för experimentet i detta arbete är en vidareutveckling av Hawkins(2017) källkod baserad på Harris(2007) som implementeras i Unity med hjälp av ett MonoBehaviour-script som styr körningen av ett antal HLSL-shaders vilka motsvarar olika delsteg i simuleringen. Simuleringen utvecklas med avsikt att efterlikna den effekt som presenteras i Guay, Colin och Egli (2011) där fokus ligger på att upprepa den övergripande metoden för att rendera eld i screen-space med hjälp av en Eulersk vätskesimulering. Simuleringen i detta arbete motsvarar inte den i Guay, Colin och Egli (2011) gällande de exakta algoritmer som används för vätskesimuleringen då själva vätskesimuleringen i detta arbete istället bygger på Harris(2007). Detta arbete är därför snarast att betrakta som en upprepning av det koncept som Guay, Colin och Egli (2011) beskriver och en visuell bedömning görs under implementeringen för att så nära som möjligt efterlikna det önskade utseendet på elden.

Experimentet i detta arbete genomförs genom att för ett antal olika upplösningar på eldsimuleringen mäta beräkningstiden och spara detta som datapunkter. I Guay, Colin och Egli (2011) syns hur upplösningen har en stor påverkan på uppdateringsfrekvensen. Detta kan tänkas bero på en ökad beräkningstid vid högre upplösningar och detta är skälet till att upplösning valts som en oberoende variabel för experimentet i detta arbete. För varje upplösning kommer också mängden iterationer i det simuleringssteg som beräknar tryckgradienten genom Jacobi-metoden varieras. I Harris (2007) beskrivs hur mängden Jacobi-iterationer är avgörande för en korrekt simulering men eftersom det är en iterativ process har mängden iterationer också en avsevärd effekt på simuleringens hastighet. Detta motiverar varför mängden Jacobi-iterationer har valts som ytterligare en oberoende variabel i experimentet.

För varje konfiguration på simuleringen avseende upplösning samt mängden Jacobi-iterationer körs simuleringen tills 10 000 uppdateringscykler passerat varefter den genomsnittliga beräkningstiden för dessa cykler samlas in. Varje konfiguration mäts alltså med 10 000 samplingar. Eliasson (2019) tar upp hur reliabilitet är en viktig faktor i kvantitativa undersökningar. För att stärka reliabiliteten redogörs också för hur resultatet måste kunna upprepas för att kunna kontrollera insamlad data i en kvantitativ undersökning. Det nämns även hur validitet är en viktig faktor och att validiteten kan stärkas genom att säkerställa att mätningen sker på rätt data. Som nämnts tidigare kommer varje mätning ske över 10 000 uppdateringscykler. Detta är ett sätt att stärka reliabiliteten då det jämnar ut insamlad mätdata. Det är också ett sätt att stärka validiteten då det minskar påverkan på resultatet från faktorer utanför experimentmiljön, som till exempel andra datorprocesser.

Det kan argumenteras för att en jämförelse mellan resultaten i detta arbete och resultaten i Guay, Colin och Egli (2011) hade kunnat ge intressanta insikter gällande simuleringsmetodens prestanda i stort. Dock finns det ett flertal bakomliggande variabler som till exempel olika verktyg för prestandamätning eller olika utvecklingsmiljöer som kan störa jämförelsen och därför kommer resultaten från experimentet i detta arbete att analyseras oberoende från resultaten som presenteras i Guay, Colin och Egli (2011).

3.3 Artefakt

En artefakt utvecklades i Unity i syfte att fungera som experimentmiljö. Artefakten består av

en scen som konfigurerats för att efterlikna den metod för att rendera eld som beskrivs i Guay, Colin och Egli (2011). Eftersom det inte framgår i detalj hur renderingen fungerar i Guay, Colin och Egli (2011) gjordes en tolkning av processen som anpassades för att fungera i Unity. De centrala delarna av processen som identifierats samt också återfinns i Unity-projektet är:

- Rastrering av 3D-objekt till en textur som anger var eld skall uppstå i simuleringen.
- Eulersk vätskesimulering där tryckgradienten skalas upp.
- Rendering av densitetsfält tillsammans med 3D-scen där densitetsvärdet renderas som eldsflammor 'ovanpå' 3D-miljön.

På grund av skillnader mellan utvecklingsmiljön i Guay, Colin och Egli (2011) och utvecklingsmiljön i detta arbete uppstod svårigheter vid implementering av den exakta metod för själva vätskesimuleringen som beskrivs i deras arbete. På grund av detta valdes en annan metod för vätskesimulering som bedömdes kunna modifieras för att uppnå samma effekt som i Guay, Colin och Egli (2011). Den vätskesimulering som utvecklades för artefakten är därför en vidareutveckling av en Unity-implementation som utvecklats av Hawkins (2017) vilken i sin tur bygger på den metod som beskrivs i Harris(2007). Vätskesimuleringen anpassades för att så nära som möjligt efterlikna effekten i Guay, Colin och Egli (2011) sett till både den visuella effekten samt den övergripande tekniska metoden för att uppnå effekten.

En mängd modifieringar av källkoden från Hawkins(2017) fick göras för att uppnå den effekt som önskades. Ändringarna omfattar bland annat en lösning för att ge simuleringen möjlighet att läsa av de texturer som rastreras med värden för var densitet och temperatur skall adderas samt implementering av en shader som adderar nämnda kvantiteter. Ändringar behövde också göras kring hur renderingen av den slutgiltiga texturen vid varje uppdateringscykel sker där stor vikt lades vid att ge rätt färgvärden samt intensitet för de resulterande eldsflammorna. Utöver detta gick en större del av tiden till att modifiera de olika ingångsvärden i eldsimuleringen för att få den slutgiltiga renderingen att se ut som en trovärdig eld i stil med Guay, Colin och Egli (2011). Detta innefattar bland annat att se till så att simuleringen tillåter ett visst mått av divergens för att som beskrivs i Guay, Colin och Egli (2011) skapa toppar och dalar i skalärfältet för tryck vilket leder till att eldsflammorna får ett mer turbulent beteende och lättare 'bryts upp' vilket också ger den illusion av rörelse i djupled som skiljer denna teknik från andra tvådimensionella eldsimuleringar.

Valet av Unity som utvecklingsmiljö motiveras av att det är den miljö vi är mest bekanta med. Dessutom är det den spelmotor i vilken Hawkins(2017) metod är utvecklad. Toftedahl och Engström(2019) presenterar i en sammanställning av data hämtad från Steam den 20 december 2018, att Unity är den näst mest använda spelmotorn bland de undersökta speltitlarna. Unity används som spelmotor i 13,2% av titlarna. Detta kan ses som en indikation på att Unity är en populär spelmotor vilket vidare motiverar valet av Unity som spelmotor för experimentet i vårt arbete.

3.4 Datainsamling

En lösning för att mäta tidsåtgången för ett simuleringssteg på grafikkortet implementerades med hjälp av Unitys Recorder-API för mätning av GPU-tid, Recorder.gpuElapsedNanoseconds (Unity Technologies (2024). Markörer för tidsmätning placerades i Update-loopen i det MonoBehaviour-script som styr simuleringen med hjälp av

Unitys CustomSampler-klass (Unity Technologies (2024).

För själva mätningen av tidsåtgången på GPU:n skapades 28 stycken ‘development builds’ av projektet med olika konfigurationer gällande upplösning på eldsimuleringen samt mängd Jacobi-iterationer. Upplösningarna valdes baserat på vad som bedömdes vara vanliga upplösningar för ett bildförhållande om 16:9 och mängderna iterationer för Jacobi-steget valdes inom ett spann för vad som bedömdes ge en acceptabel visuell kvalitet på eldsimuleringen. Upplösningen på simuleringen är oberoende från upplösningen på själva applikationen.

Upplösningar: *320x180, 640x360, 960x540, 1280x720, 1920x1080, 2560x1440, 3840x2160*

Jacobi-iterationer: *20, 35, 50, 65*

När 10 000 tidsmätningar vardera gjorts under körning av respektive build beräknades medelvärdet automatiskt i MonoBehaviour-scriptet och presenterades på skärmen varpå detta värdet noterades manuellt. Detta upprepades för varje build. När alla mätvärden samlats in sammanställdes resultatet i en samling diagram för analys.

3.5 Metoddiskussion

Det kan argumenteras för att en brist med den valda metoden är att endast ett medelvärde samlats in. Det finns en risk att intressant information försvinner när en mängd datapunkter sammanställs på detta sätt. Exempel på information som kunde samlats in är högsta respektive lägsta uppmätta tid per konfiguration, eller information gällande ojämnheter i uppmätt tid mellan olika uppdateringscykler, i form av någon typ av avvikelsedata.

Orsaken bakom valet av att endast samla in medelvärdet är att experimentet föregicks av en grundlig undersökning, där det med hjälp av Unitys Profiler-verktyg framgick att tidsåtgången för varje uppdateringscykel på GPU:n var, enligt vår bedömning, jämn. Undersökningen var också menad som ett sätt att säkerställa att den tid vi samlade in med hjälp av Unitys Recorder-API verkligen var rätt tid. Vi jämförde alltså den tid som mättes genom API:n med den tid som presenterades i Unitys profiler-verktyg.

Trots detta hade det rimligtvis varit fördelaktigt för arbetet som helhet att samla in och redovisa i större detalj hur eldsimuleringen presterade. Detta är något som med fördel kan undersökas vidare i framtida forskning på ämnet. I detta arbete blev själva experimentet något begränsat till följd av den tid som gick åt för att implementera själva simuleringen på ett tillfredsställande vis. Därför valdes medelvärdet som den typ av data som ansågs bäst kunna svara på frågeställningen.

4 Genomförande och analys

4.1 Artefakt

En artefakt utvecklades i Unity 2022.3.12f1 där en scen utformades för att köra eldsimuleringen samt mäta prestandan. Artefakten är en vidareutveckling av Hawkins(2017) Unity-projekt. Hawkins metod för vätskesimuleringen är i korthet uppbyggt på så vis att ett monobehaviour-script hanterar dataflödet till en serie shaders. Dessa shaders utför beräkningarna som krävs för att simulera fluiden i fråga.

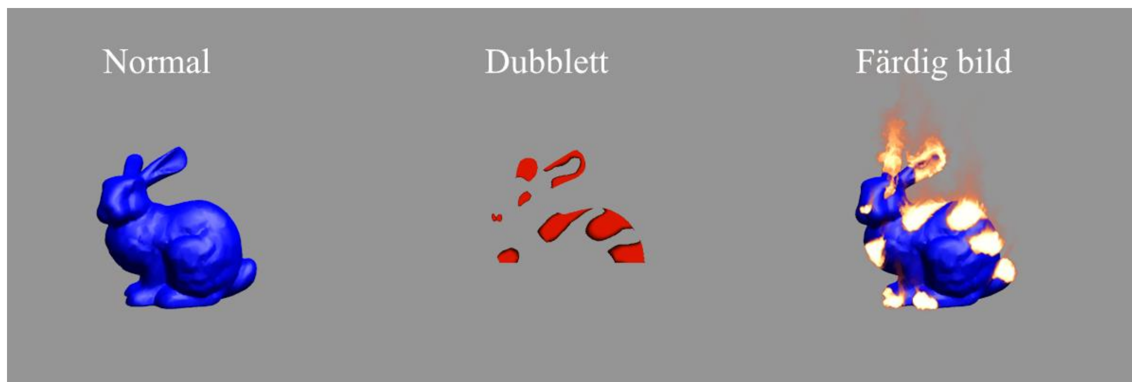
En potentiell nackdel med den här lösningen är de många Graphics.Blit-operationerna som krävs för att skicka data mellan CPU:n och GPU:n. Hawkins påpekar hur detta är något som innebär mycket arbete för GPU:n. Anledningen till de många Graphics.Blit-operationerna är sannolikt på grund av att shader-koden är parallelliserad vilket gör det olämpligt att till exempel läsa från och skriva till en och samma textur samtidigt. I Hawkins(2017) lösning behöver ofta samma textur kunna läsas från och skrivas till samtidigt och lösningen i detta fallet är att dessa texturer finns i två exemplar.

Det är möjligt att simuleringen hade kunnat optimeras genom att försöka minimera antalet Graphics.Blit-operationer. Detta bedömdes dock vara en större modifikation på vätskesimuleringens arkitektur vilket inte passade inom arbetets tidsram. Därför valde vi att endast modifiera Hawkins lösning till den grad som krävdes för att efterlikna metoden som presenteras av Guay, Colin och Egli (2011).

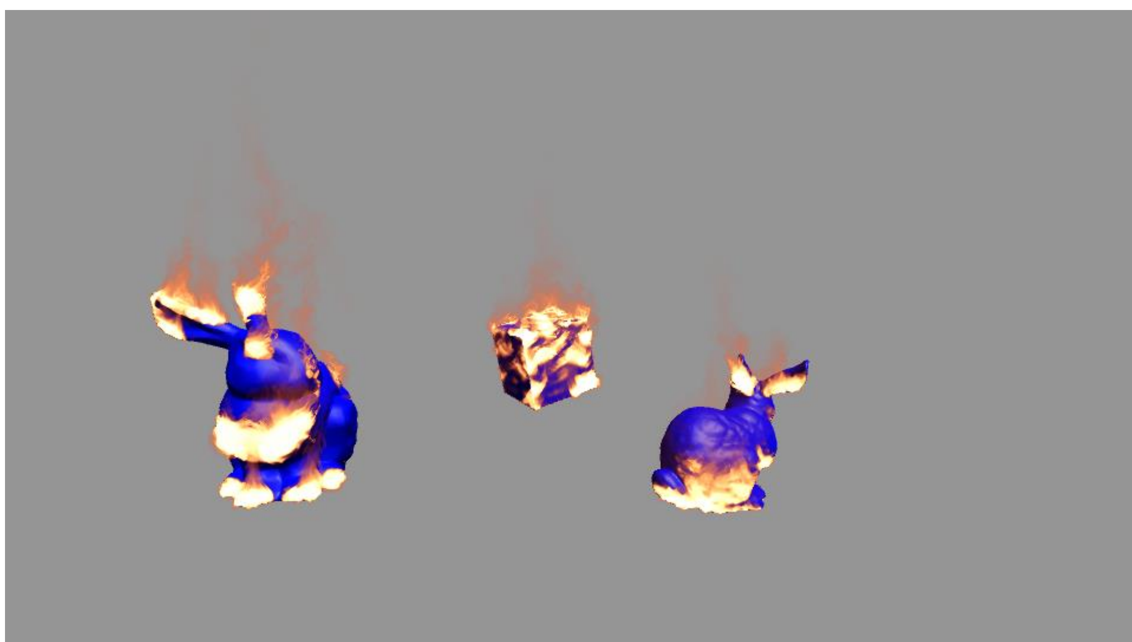
4.1.1 Unity-scen

Scenen består av två kameror och ett antal 3D-objekt samt ett spelobjekt som huserar det MonoBehaviour-script som styr eldsimuleringen. 3D-objekten i scenen är två stycken kaniner och en kub, vilket kan ses i figur 4 och figur 6. Alla 3D-objekt som skall sättas eld på i simuleringen har en dubblett. Dubbletterna huserar på ett eget lager i scenen för att kunna särskilja dessa från övriga objekt vid rastrering. Dubbletterna är försedda med egna material där texturen på materialet utformats för att ge ett lämpligt färgvärde vid rastrering, vilket illustreras i figur 5. 3D-objekten, inklusive dubbletterna, har tillsammans 78 900 trianglar. Två kameror behövs för att kunna rastrera scenen på två olika sätt. Den ena kameran rastrerar alla objekt i vanlig ordning förutom dubbletterna som utesluts ur denna rastrering. Den andra kameran rastrerar endast dubbletterna och sparar bilden till en textur som är åtkomlig för eldsimuleringen. I eldsimuleringen adderas densitet och temperatur till simuleringen för varje pixel i simuleringen baserat på färgvärdet hos motsvarande pixel i den rastrerade texturen med dubbletter.

Figur 4: I scenen finns två kameror samt två kaniner och en kub. Kaninerna och kuben finns alla i två exemplar.



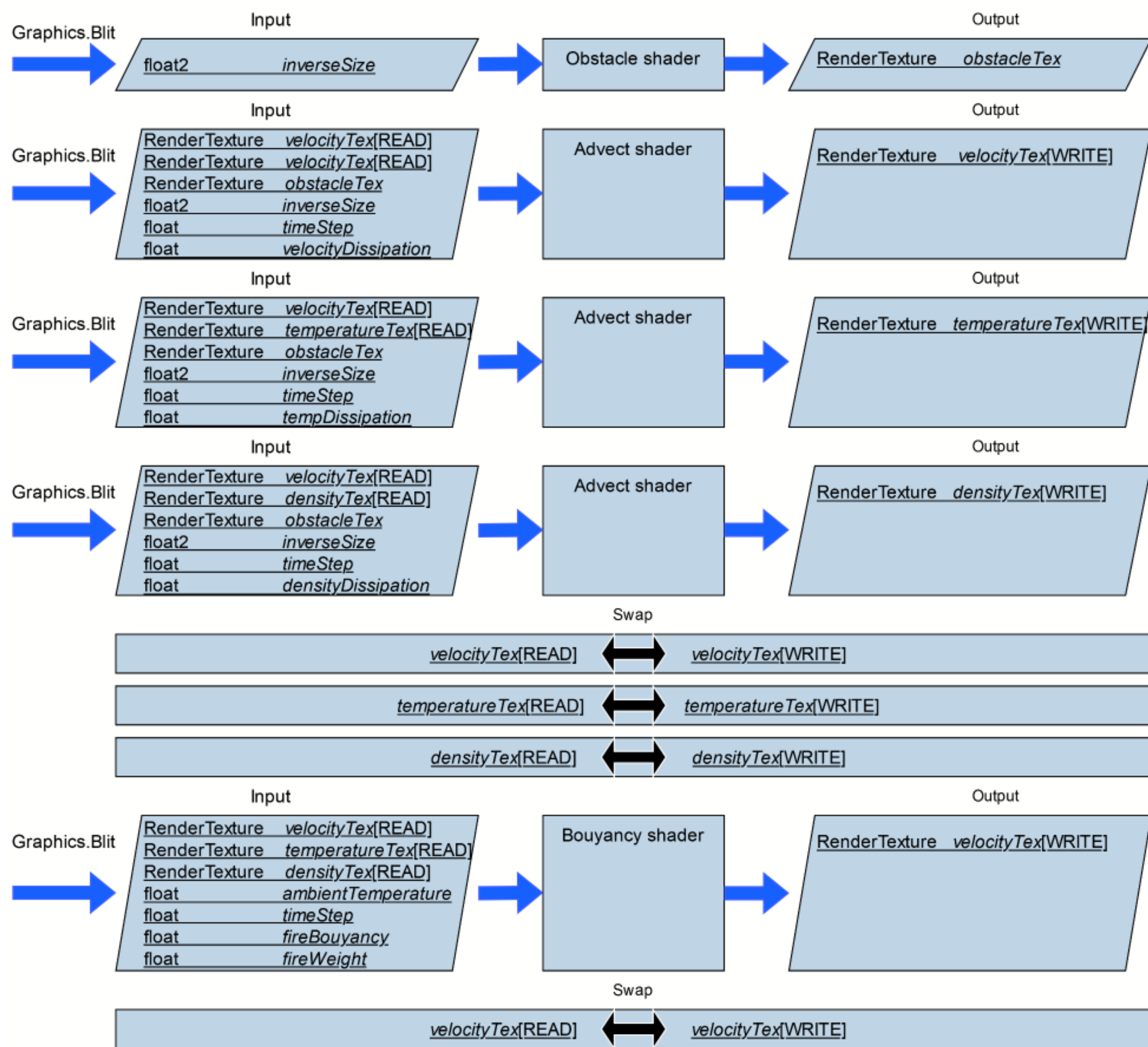
Figur 5: Sammansatt visualisering av de två kamerornas rastrerade bilder samt slutgiltig eldsimulering. Obs: Bilden är alltså inte representativ för hur scenen ser ut utan är till för att visa skillnaden mellan de rastrerade objekten samt deras effekt på eldsimuleringen. Normal = modell, Dubblett = modell med material för 'bränsle', Färdig bild = slutlig rendering.



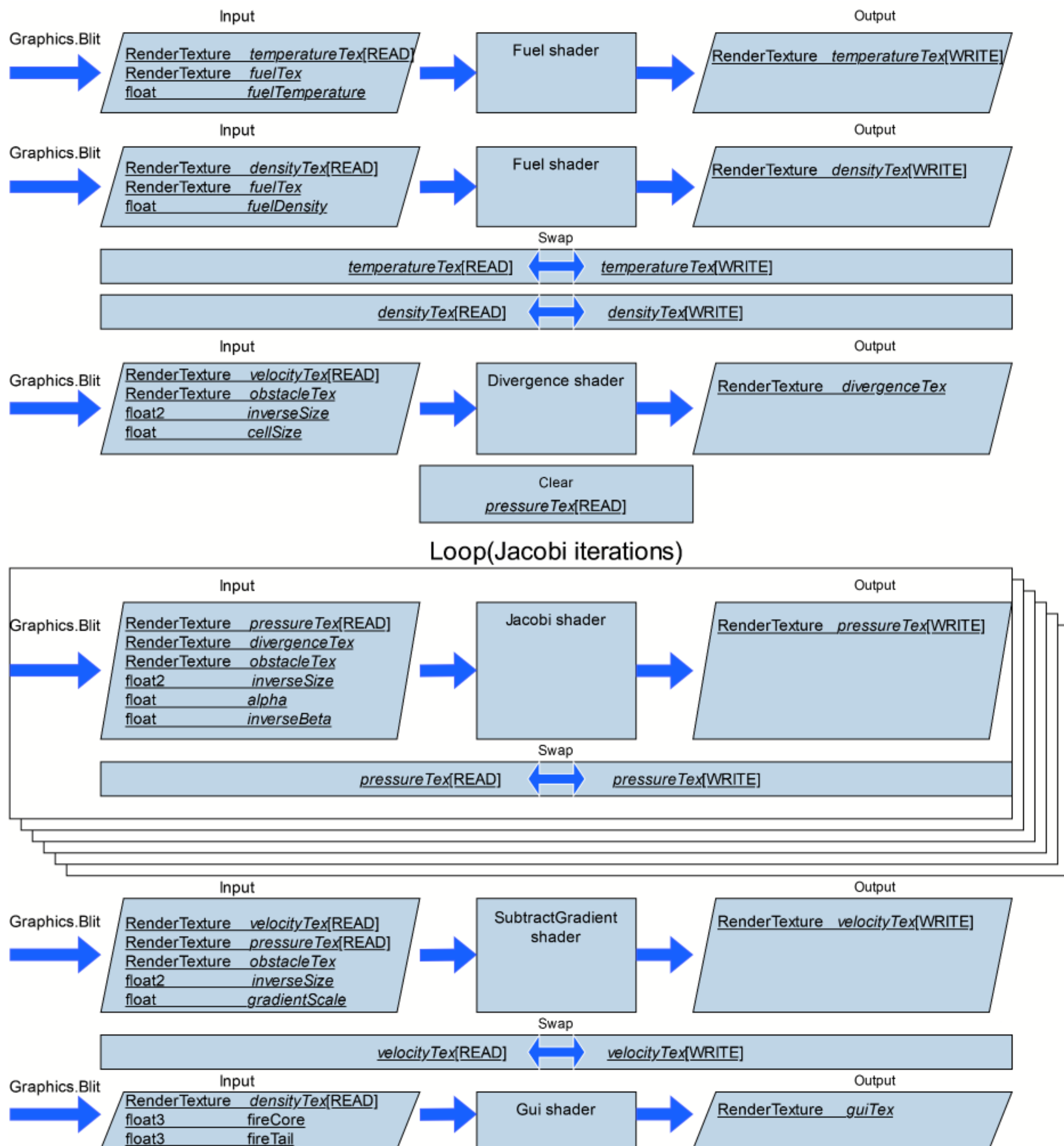
Figur 6: Bildruta ur eldsimulering med upplösning 960 x 540 samt 50 Jacobi-iterationer. Två kaniner samt en kub.

4.1.2 MonoBehaviour

Eldsimuleringen består av ett C#-script som styr ett antal HLSL-shaders vilka beräknar olika delsteg i simuleringen. Simuleringsscriptet har referenser till de material som används för att kunna köra shaderkoden med hjälp av Graphics.Blit-funktionen i Unity. Varje material svarar mot en shader. Varje shader svarar mot ett delsteg i simuleringen. För att lagra värden mellan uppdateringcykler används texturer i form av Unitys RenderTexture. Simuleringsscriptet har i uppgift att sköta ordningen i vilken de olika delstegen i simuleringen körs samt att se till att varje shader har tillgång till rätt texturer och övriga variabler. I figur 7 och figur 8 illustreras hur varje shader används i C#-scriptet under en uppdateringscykel.



Figur 7: Flödesschema över de första fem Graphics.Blit-operationerna under en Update-cykel. Graphics.Blit kallas från ett Monobehaviour-script. Schemat läses uppifrån och ner. Varje rad visar en Graphics.Blit-operation med in- och utdata, alternativt en swap-operation där två texturer byter plats.

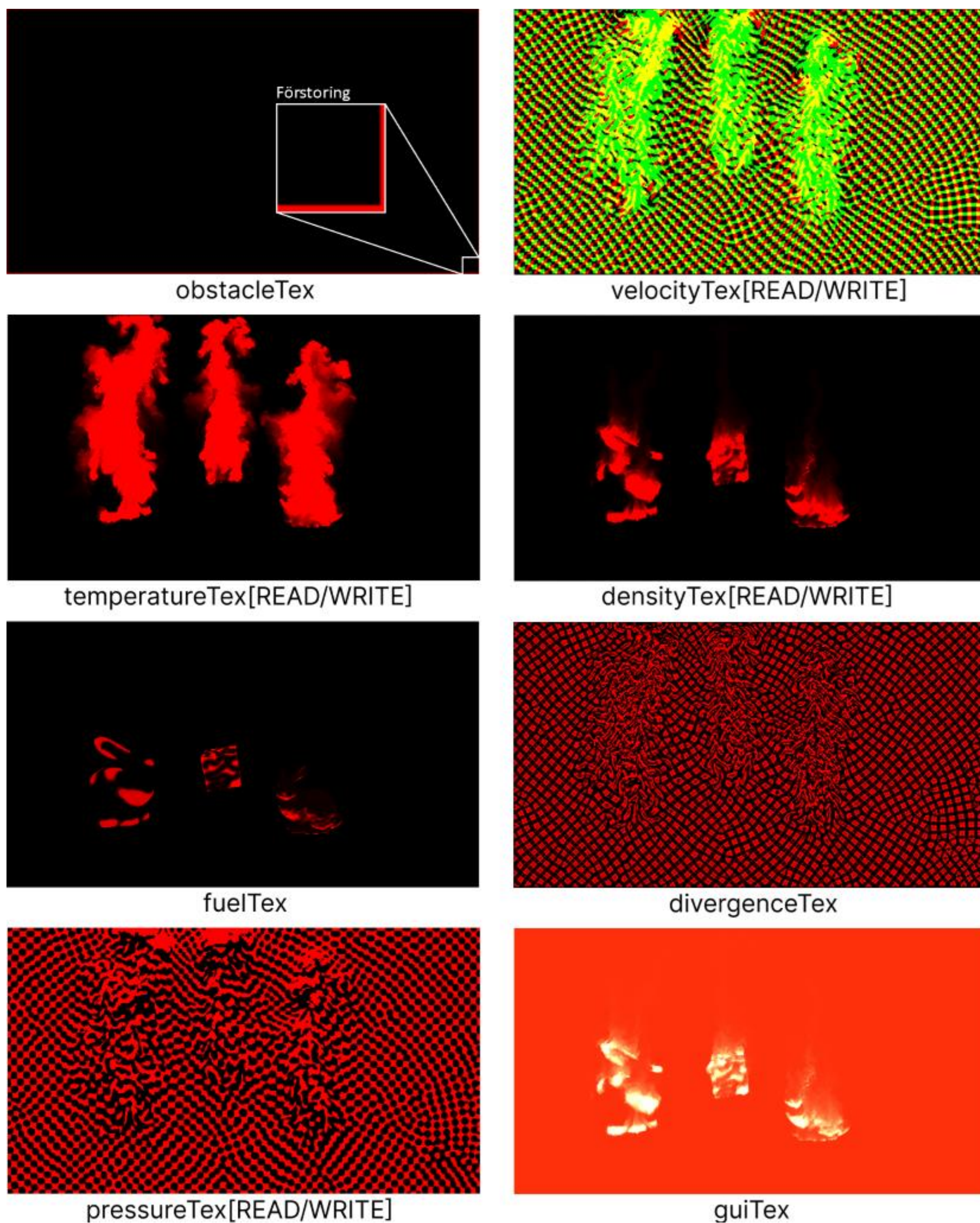


Figur 8: Flödesschema över de efter figur 7 följande Graphics.Blit-operationerna under en Update-cykel. Graphics.Blit kallas från ett Monobehaviour-script. Schemat läses uppifrån och ner. Varje rad visar en Graphics.Blit-operation med in- och utdata alternativt en swap-operation där två texturer byter plats. Jacobi-shadern körs genom att i en loop i Monobehaviour-scriptet kalla på Graphics.Blit.

4.1.3 Texturer

För simuleringen krävs att värden lagras för olika egenskaper hos fluiden vid varje uppdateringscykel samt mellan cykler. Detta görs med hjälp av texturer i form av Unitys RenderTexture-klass vilka kan manipuleras med hjälp av de shaders som används i simuleringen. Texturerna motsvarar de ‘rutnät’ eller ‘fält’ som beskrivits tidigare och är centrala för att kunna implementera en Eulersk vätskesimulering. I figur 9 illustreras hur texturerna kan se ut under en uppdateringscykel.

- *obstaclesTex - skalära värden för hinder*
- *2x velocityTex - (read/write) tvådimensionella vektorer för rörelse*
- *2x temperatureTex - (read/write) skalära värden för temperatur*
- *2x densityTex - (read/write) skalära värden för densitet*
- *fuelTexture - skalära värden för rastrerat 'bränsle'*
- *divergenceTex - skalära värden för divergens*
- *2x pressureTex - (read/write) skalära värden för tryckgradient*
- *guiTex - färgvärde för den slutgiltiga eldeffekten*



Figur 9: Exempel på hur de texturer som används i simuleringen kan se ut under en uppdateringscykel. Obs: velocityTex, densityTex, pressureTex och temperatureTex finns alla i två exemplar för att kunna läsas från och skrivas till samtidigt under simuleringen [READ/WRITE].

4.1.4 Shaders

Nedan följer en redovisning av de shaders som används för simuleringen. Dessa är skrivna i HLSL och kallas genom Update-loopen i MonoBehaviour-scriptet med hjälp av Graphics.Blit där resulterande output-värden sparas i relevanta texturer. Endast fragment-delen av varje shader används för simuleringen och därför redovisas endast denna del.

Obstacle

I Hawkins(2017) lösning uppdaterar Obstacle-shadern en textur som lagrar var i screen-space hinder skall finnas. Ett hinder påverkar vätskesimuleringen i senare steg genom att fluiden inte kan befinna sig vid berörda pixlar. I simuleringen för experimentet är endast pixlarna i bildens ytterkanter markerade som hinder.

I experimentet i detta arbete användes Hawkins(2017) lösning, vilken kan ses i figur 10, med undantag för att vi plockat bort kod som användes för att rita ett cirkelformat hinder. Vi har endast behållit den del som ritar hinder runt bildens kanter.

```
float2 inverseSize // = (simuleringens upplösning)/1, på vardera axel

float4 fragment()
{
       float4 result = (0, 0, 0, 0);

       if (UV.x <= inverseSize.x)
           result = (1, 1, 1, 1);

       if (UV.x >= 1.0 - inverseSize.x)
           result = (1, 1, 1, 1);

       if (UV.y <= inverseSize.y)
           result = (1, 1, 1, 1);

       if (UV.y >= 1.0 - inverseSize.y)
           result = (1, 1, 1, 1);

       return result;
}
```

Figur 10: Fragment-shader för obstacle-steget. Output: RenderTexture obstacleTex.

Advect

Advect-shadern ansvarar för advektion av olika egenskaper i fluiden vilket motsvarar advektionstermen i Navier-Stokes ekvation för icke-komprimerbara Newtonska fluider. Detta beskrivs av Harris(2007) som en förflyttning av kvantiteter för olika egenskaper mellan rutor över tid. Harris nämner också att eftersom simuleringen uppdateras med hjälp av ett diskret rutnät, en pixel i en textur är en ruta, kan advektion av kvantiteter i fluiden implicit beräknas genom att för respektive pixel i texturen spåra kvantitetens position i rutnätet bakåt i tiden enligt δt och u. Detta enligt ekvation 19 där u är velociteten vid koordinaten x och δt är tidssteget. q är den kvantitet som skall förflyttas och t är nuvarande tid. Därefter kan man kopiera den önskade kvantiteten q från denna spårade position till den aktuella pixeln.

$$q(x, t + \delta t) = q(x - u(x,t)\delta t, t)$$

Ekvation 19: Ekvation för implicit advektion - Harris(2007)

Shadern används i flera steg i simuleringen för att förflytta kvantiteterna velocitet, densitet eller temperatur för den aktuella pixeln enligt 2D-vektorn för velocitet vid den aktuella pixeln som lagras i velocitetsfältet. Shadern matas varje gång med velocitetstexturen samt vid respektive steg den textur som lagrar värden för kvantiteten som skall förflyttas, till exempel densitetstexturen. Både skalärvärden och vektorer kan förflyttas i denna shader.

I experimentet har Hawkins(2017) metod, vilken kan ses i figur 11, för advektion använts vilken i sin tur bygger på den implicita metod som presenteras av Harris(2007). I Hawkins metod multipliceras den förflyttade kvantiteten också med ett dämpningsvärde för att simulera hur kvantiteten 'sprids ut' vid förflyttning. Hawkins har dessutom introducerat ett steg för att eliminera kvantiteter vid koordinater som markerats som hinder i obstacleTex.

```
sampler2D velocityTex[READ]
sampler2D densityTex[READ]*
sampler2D obstacleTex
float2 inverseSize // = (simuleringens upplösning)/1, på vardera axel
float timeStep
float densityDissipation**

float4 fragment()
{
	float2 u = tex2D(velocityTex[READ], UV).xy;

	float2 coord = UV - (u * inverseSize * timeStep);

	float4 result = densityDissipation* tex2D(densityTex[READ], coord);

	float solid = tex2D(obstacleTex, UV).x;

	if(solid > 0.0) result = float4(0,0,0,0);

	return result;
}
```

Figur 11: Fragment-shader för advect-steget. Output: RenderTexture densityTex[WRITE]*

*kan också vara velocityTex[READ] eller temperatureTex[READ]

**kan också vara velocityDissipation eller tempDissipation

Buoyancy

Bouyancy-shadern beräknar fluidens flytkraft vid den aktuella pixeln genom att jämföra temperaturen från skalärfältet för temperatur med den omgivande temperaturen som lagras i ett konstant skalärt värde i MonoBehaviour-scriptet. Detta relaterar till den fjärde och sista termen **F**, i Navier-Stokes ekvation för icke-komprimerbara Newtonska fluider, vilken beskriver externa krafter. Flytkraften är alltså en extern kraft i sammanhanget. Flytkraften skrivs till aktuell pixel i vektorfältet för velocitet enligt ekvation 20, där T är temperaturen för fluiden och T_0 den omgivande temperaturen medan $\hat{\jmath}$ representerar riktningen för den

resulterande flytkraften vilken pekar uppåt för denna simulering (Harris 2007).

$$f_{buoy} = \sigma(T - T_0)\hat{j}$$

Ekvation 20: Ekvation för flytkraft - Harris(2007)

I experimentet användes Hawkins(2017) lösning, vilken kan ses i figur 12, för flytkraft vilken bygger på Harris(2007) beskrivning av fenomenet. Hawkins lösning adderar också vikt för fluiden vilket motverkar flytkraften. Därför motsvarar σ i det här fallet resultatet av både flytkraft och vikt hos fluiden.

```
sampler2D velocityTex[READ]
sampler2D temperatureTex[READ]
sampler2D densityTex[READ]
float ambientTemperature
float timeStep
float fireBouyancy
float fireWeight

float4 fragment()
{
	float T = tex2D(temperatureTex[READ], UV).x;
	float2 V = tex2D(velocityTex[READ], UV).xy;
	float D = tex2D(densityTex[READ], UV).x;
	float2 result = V;

	if(T > ambientTemperature)
	{
		result += (timeStep * (T - ambientTemperature) * fireBouyancy
		- D * fireWeight) * float2(0, 1);
	}

	return float4(result, 0, 1);
}
```

Figur 12: Fragment-shader för buoyancy-steget. Output: RenderTexture velocityTex[WRITE]

Fuel

Fuel-shadern, vilken kan ses i figur 13, är en central del i simuleringen då den baserat på aktuell kameravy beräknar hur mycket eld som skall adderas i simuleringen. Denna shader är utvecklad specifikt för detta arbete med metoden som presenteras i Guay, Colin och Egli (2011) som inspiration. Elden adderas genom att öka densitet samt temperatur vid berörda pixlar. Endast ett av värdena adderas vid varje körning av shadern och därför körs den en gång för densitet och en gång för temperatur vid varje uppdateringscykel.

Shadern läser det skalära färgvärdet vid den aktuella pixeln från texturen fuelTex vilken lagrar färgvärdena från rastreringen av de objekt vars material skall fungera som bränsle för simuleringen och multiplicerar detta med en konstant faktor för hur mycket av önskad kvantitet som skall adderas i detta steg. Kvantiteten uppdateras på den aktuella pixeln i texturen som lagrar värden för den berörda egenskapen.

```
sampler2D temperatureTex[READ]*
```

```
sampler2D fuelTex
float fill**

float4 fragment()
{
	float impulse = 0;
	float fuel = tex2D(fuelTex, UV).x;
	if(fuel > 0)
	{
		impulse = fuel;
	}
	float source = tex2D(temperatureTex[READ], UV).x;
	return max(0, lerp(source, fill, impulse)).xxxx;
}
```

Figur 13: Fragment-shader för fuel-steget. Output: RenderTexture temperatureTex[WRITE]***

*kan också vara densityTex[READ]

**fuelTemperature eller fuelDensity

***kan också vara densityTex[WRITE]

Divergence

Divergence-shadern utför det första av tre steg för att utföra projektionsberäkningen i simuleringen vilket, som Harris(2007) beskriver, har i syfte att se till att kriteriet för icke-komprimerbarhet uppfylls i simuleringen. Projektionsberäkningen, med dess tre steg, kan ses som ett delsteg i den övergripande processen. Projektionsberäkningen innebär att man i första steget beräknar divergensen i velocitetsfältet. Det är detta som görs i Divergence-shadern. Detta första steg är endast till för att beräkna en temporär textur som skall användas i följande steg för projektionsberäkningen. Steg två innebär att man med hjälp av den beräknade divergensen löser Poisson-ekvationen för att beräkna tryckgradienten, vilket görs i Jacobi-shadern. Det tredje steget innebär att man subtraherar tryckgradienten från velocitetsfältet, vilket resulterar i ett divergensfritt velocitetsfält som därmed uppfyller villkoret för icke-komprimerbarhet (Harris 2007).

$$\nabla \cdot u = 0$$

Ekvation 21: Navier-Stokes kontinuitetsekvation vilken beskriver icke-komprimerbarhet. - Harris(2007)

$$\nabla \cdot u = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

Ekvation 22: Definition av divergens - Harris(2007)

$$\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\delta y}$$

Ekvation 23: Differensoperatorform av divergensekvation - Harris(2007)

Shadern läser velocitet samt hinder i kringliggande pixlar runt aktuell pixel och beräknar divergensen vilken lagras i den temporära texturen. Divergensen avser nettoflödet ut från

den aktuella pixeln och en textur med värden för divergens är avgörande för att kunna beräkna tryckgradienten för simuleringen (Harris 2007).

I experimentet används Hawkins(2017) lösning, vilken kan ses i figur 14, för beräkning av divergensen hos velocitetsfältet. Hawkins metod tar också hänsyn till de hinder som definierats i obstacleTex.

```
sampler2D velocityTex[READ]
sampler2D obstacleTex
float2 inverseSize
float cellSize

float4 fragment()
{
// Find neighboring velocities:
float2 vN = tex2D(velocityTex[READ], UV + float2(0, inverseSize.y)).xy;
float2 vS = tex2D(velocityTex[READ], UV + float2(0, - inverseSize.y)).xy;
float2 vE = tex2D(velocityTex[READ], UV + float2(inverseSize.x, 0)).xy;
float2 vW = tex2D(velocityTex[READ], UV + float2(- inverseSize.x, 0)).xy;

// Find neighboring obstacles:
float bN = tex2D(obstacleTex, UV + float2(0, inverseSize.y)).x;
float bS = tex2D(obstacleTex, UV + float2(0, -inverseSize.y)).x;
float bE = tex2D(obstacleTex, UV + float2(inverseSize.x, 0)).x;
float bW = tex2D(obstacleTex, UV + float2(-inverseSize.x, 0)).x;

// Set velocities to 0 for solid cells:
if(bN > 0.0) vN = 0.0;
if(bS > 0.0) vS = 0.0;
if(bE > 0.0) vE = 0.0;
if(bW > 0.0) vW = 0.0;
float result = cellSize * (vE.x - vW.x + vN.y - vS.y);

return float4(result,0,0,1);
}
```

Figur 14: Fragment-shader för divergence-steget. output: RenderTexture divergenceTex

Jacobi

Jacobi-shadern utför steg två i projektionsberäkningen och beräknar med hjälp av Jacobi-metoden som beskrivs i Harris(2007) värden för en textur där tryckvärdena i texturen skall uppfylla villkoret för icke-komprimerbarhet. Detta genom att iterativt lösa Poisson-ekvationen avseende divergens. Detta genom att ‘gissa’ vad lösningen är och sedan för varje iteration använda resultaten från föregående iteration för att beräkna nästa resultat. Antalet iterationer påverkar hur väl villkoret för icke-komprimerbarhet uppfylls där fler iterationer ger ett resultat som är mer fysiskt korrekt. Resulatet skrivs till en textur som i nästa steg kommer användas för att subrahera tryckgradienten från velocitetsfältet. Detta för att minimera divergensen.

$$\nabla^2 p = \nabla \cdot w$$

Ekvation 24: Poisson-ekvation för tryck, härledd från Helmholtz-Hodge-dekompositionen, där $\nabla^2 p$ är

laplaceoperatorn och $\nabla \cdot w$ är divergensen hos velocitetsfältet. - Harris(2007)

$$p_{i,j}^{k+1} = \frac{1}{4}(p_{i-1,j}^{k} + p_{i+1,j}^{k} + p_{i,j-1}^{k} + p_{i,j+1}^{k} - \alpha \cdot \nabla \cdot w_{i,j})$$

Ekvation 25: Diskretiserad Poissonekvation för tryck

Poisson-ekvationen för tryck löses, som Harris(2007) beskriver, med hjälp av Jacobi-metoden genom att för varje ruta simuleringen applicera ekvation 25, den diskretiserade Poisson-ekvationen, och uppdatera värdet för tryck enligt denna. Eftersom Jacobi-shadern endast uppdaterar trycket en gång behöver den loopas från Monobehaviour-scriptet enligt den förutbestämda mängden Jacobi-iterationer. Vid varje iteration används det tryckvärde som beräknats i föregående iteration i ekvationen. Detta leder till att varje iteration förändrar trycket mindre och mindre då lösningen närmar ett tillstånd där den konvergerar. Dock kommer lösningen i praktiken inte att konvergera. Detta på grund av den förutbestämda mängden iterationer. Av detta följer att ju fler iterationer Jacobi-shadern körs, desto närmare kommer simuleringen ett tillstånd där villkoret för icke-komprimerbarhet uppfylls. När Jacobi-shadern körts så många iterationer som bestämts kommer den resulterande texturen pressureTex, tryckgradienten, användas i SubtractGradient-shadern för att subtrahera denna från velocitetsfältet vilket i sin tur leder till att simuleringen uppfyller villkoret för icke-komprimerbarhet till den grad som antalet Jacobi-iterationer tillåter.

I experimentet används den implementation av Jacobi-metoden som Hawkins(2017) utvecklat baserat på Harris(2007) metod. Denna kan ses i figur 15. Hawkins metod tar också hänsyn till de hinder som definierats i obstacleTex.

```
sampler2D pressureTex[READ]
sampler2D divergenceTex
sampler2D obstacleTex
float2 inverseSize
float alpha
float inverseBeta

float4 fragment()
{
	// Find neighboring pressure:
	float pN = tex2D(pressureTex[READ], UV + float2(0,
	inverseSize.y)).x;
	float pS = tex2D(pressureTex[READ], UV + float2(0, -
	inverseSize.y)).x;
	float pE = tex2D(pressureTex[READ], UV + float2(inverseSize.x,
	0)).x;
	float pW = tex2D(pressureTex[READ], UV + float2(-inverseSize.x,
	0)).x;
	float pC = tex2D(pressureTex[READ], UV).x;

	// Find neighboring obstacles:
	float bN = tex2D(obstacleTex, UV + float2(0, inverseSize.y)).x;
	float bS = tex2D(obstacleTex, UV + float2(0, -inverseSize.y)).x;
	float bE = tex2D(obstacleTex, UV + float2(inverseSize.x, 0)).x;
	float bW = tex2D(obstacleTex, UV + float2(-inverseSize.x, 0)).x;
```

```
	// Use center pressure for solid cells:
	if(bN > 0.0) pN = pC;
	if(bS > 0.0) pS = pC;
	if(bE > 0.0) pE = pC;
	if(bW > 0.0) pW = pC;
	float bC = tex2D(divergenceTex, UV).x;
	return (pW + pE + pS + pN + alpha * bC) * inverseBeta;
}
```

Figur 15: Fragment-shader för Jacobi-steget. Output: RenderTexture pressureTex[WRITE]

SubtractGradient

$$\nabla \cdot u = 0$$

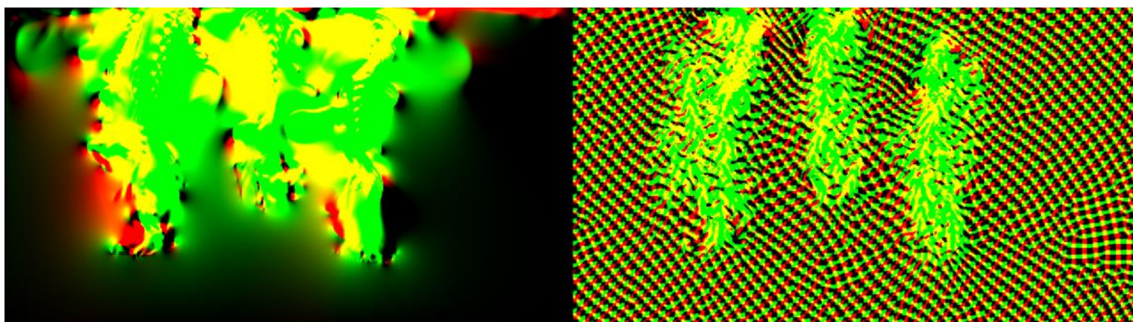
Ekvation 26: Navier-Stokes kontinuitetsekvation vilken beskriver icke-komprimerbarhet. - Harris(2007)

$$u' = u - \nabla p$$

Ekvation 27: Ekvation som beskriver hur tryckgradienten subtraheras från velocitetsfältet vilket resulterar i ett divergensfritt velocitetsfält.

SubtractGradient-shadern utför det tredje och sista steget i projektionen och läser texturen för tryckgradienten vilket anger värden för tryckförändringar där områden med högt tryck kommer röra sig mot områden med lågt tryck. Det resulterande värdet subraheras från velocitetsfältet för att justera velociteten inför nästa uppdateringscykel(Harris 2007).

Detta ser till att simuleringen uppfyller villkoret för icke-komprimerbarhet och i vanliga fall är detta vad som önskas för en stabil vätskesimulering. Dock tillåts simuleringen i detta arbete bortse från villkoret för icke-komprimerbarhet till viss del för att kunna återskapa den illusion av rörelse i djupled som beskrivs i Guay, Colin och Egli (2011). Detta genom att skala upp värdet från tryckgradienten innan det subtraheras från velocitetsfältet. I shadern är det variabeln 'gradientScale' som skalats upp. För experimentet användes ett värde av 1,3 på 'gradientScale' för att efterlikna effekten i Guay, Colin och Egli(2011). Detta värde kan sättas i inspektorn i Unity. Effekten av detta blir att de tryckvågor som uppstår vid den tryckutjämning som tryckgradienten anger förstärks till den grad att de skapar toppar och dalar i velocitetsfältet som 'kapar av' lokala flöden i vissa områden och på så vis ger en visuell effekt som kan liknas vid hur eldsflammor tycks 'brytas upp' i till exempel en brasa. I figur 16 illustreras skillnaden i velocitetsfältet vid två olika värden på 'gradientScale'. Detta är orsaken bakom varför simuleringen i Guay, Colin och Egli (2011) samt simuleringen i detta arbete kan ge illusionen av rörelse i djupled.



Figur 16: Till vänster: velocityTex vid gradientScale = 1,0 | Till höger: velocityTex vid gradientScale = 1,3

I experimentet används den implementation av divergenssubtraktionen som Hawkins(2017) utvecklat baserat på Harris(2007) metod. Denna kan ses i figur 17.

```
sampler2D velocityTex[READ]
sampler2D pressureTex[READ]
sampler2D obstacleTex
float2 inverseSize
float gradientScale

float4 fragment()
{
	// Find neighboring pressure:
	float pN = tex2D(pressureTex[READ], UV + float2(0,
	inverseSize.y)).x;
	float pS = tex2D(pressureTex[READ], UV + float2(0, -
	inverseSize.y)).x;
	float pE = tex2D(pressureTex[READ], UV + float2(inverseSize.x,
	0)).x;
	float pW = tex2D(pressureTex[READ], UV + float2(-inverseSize.x,
	0)).x;
	float pC = tex2D(pressureTex[READ], UV).x;

	// Find neighboring obstacles:
	float bN = tex2D(obstacleTex, UV + float2(0, inverseSize.y)).x;
	float bS = tex2D(obstacleTex, UV + float2(0, -inverseSize.y)).x;
	float bE = tex2D(obstacleTex, UV + float2(inverseSize.x, 0)).x;
	float bW = tex2D(obstacleTex, UV + float2(-inverseSize.x, 0)).x;

	// Use center pressure for solid cells:
	if(bN > 0.0) pN = pC;
	if(bS > 0.0) pS = pC;
	if(bE > 0.0) pE = pC;
	if(bW > 0.0) pW = pC;

	// Enforce the free-slip boundary condition:
	float2 oldV = tex2D(velocityTex[READ], UV).xy;
	float2 grad = float2(pE - pW, pN - pS) * gradientScale;
	float2 newV = oldV - grad;

	return float4(newV, 0, 1);
}
```

Figur 17: Fragment-shader för SubtractGradient-steget. Output: RenderTexture velocityTex[WRITE]

Gui

Gui-shadern sammanställer den slutgiltiga textur som skall renderas 'över' 3D-scenen. Detta genom att läsa värden från densitetsfältet och beroende på värdet skriva färgvärden till den slutgiltiga texturen. Utöver färgen skrivs också värdet för genomskinlighet baserat på densiteten. Genomskinligheten korrelerar direkt med densiteten och är noll där densiteten är noll. Färgvärdet interpoleras linjärt från två förutbestämda färgvärden baserat på densiteten där pixlar med hög densitet får en ljus, nästan vit färg och värden med lägre densitet blir mer orangea eller röda. Den resulterande texturen blir således mestadels genomskinlig med

eldsflammor på lämpliga ställen där densiteten är hög nog vilket kan härledas till vilka värden för bränslet som rastrerats.

I experimentet används en modifierad version av Hawkins(2017) lösning, vilken kan ses i figur 18. Modifikationerna gäller hur färgen hanteras och ser till att den linjära interpoleringen sker på ett sätt som ger elden önskat utseende samt ser till att texturen som skrivs till är genomskinlig där densiteten är noll.

```
sampler2D densityTex[READ]
float3 fireCore
float3 fireTail

float4 fragment()
{
	float3 col = tex2D(densityTex[READ], UV).x;
	float3 resultFire = lerp(fireTail, fireCore, col.x);
	return float4(resultFire, col.x);
}
```

Figur 18: Fragment-shader för Gui-steget. Output: RenderTexture guiTex

4.2 Experiment

För att kunna analysera prestandapåverkan av metoden som användes för eldsimuleringen i detta arbete krävdes metoder för datainsamling. Detta skedde i form av ett experiment där upplösning och mängden iterationer för Jacobi-metoden varierades för varje konfiguration av simuleringen som skulle mätas.

4.2.1 Verktyg

För att kunna mäta prestandan användes Unitys Recorder-API. Detta verktyg tillåter sampling av profileringsdata generad för ett specificerat område (Unity Technologies 2024). Detta görs i form av en sampler som finns färdig att använda om något generellt område ska analyseras. För detta arbete krävdes en anpassad sampler specifikt för att mäta påverkan på grafikkortet av de olika shaderna. Detta genomfördes genom att mäta tiden det tog för grafikkortet mellan tiden av simuleringens start och slut.

För att sedan verifiera att korrekt data samlades in användes GPU Usage-modulen i Unitys Profiler-verktyg som har funktionalitet för att mäta mängden millisekunder en process tar att utföra på grafikkortet för specifika frames (Unity Technologies 2024). Genom att sedan jämföra data insamlad via Recorder-verktyget med data mätt i Unity Profiler gick det att se att uppmätt data överensstämde tillräckligt mycket för att kunna lita på att den data som samlats in med Unitys Recorder-API är av rätt typ. Anledningen till att använda två olika metoder är att Unity Profiler inte ger möjlighet till att analysera GPU-prestanda över många frames vilket krävdes för detta arbetes datainsamling.

Alla experimenten genomfördes på en dator med följande specifikationer:

- *GPU: Nvidia GeForce RTX 3070*
- *CPU: AMD Ryzen 7 5800X, 3801 Mhz, 8 kärnor*
- *RAM: 32 GB*

4.2.2 Utförande

Genom att sätta en startpunkt för den anpassade samplern vid början av MonoBehaviour-scriptet som hanterade den övergripande funktionaliteten, som att aktivera specifika shaders, samt en slutpunkt i slutet kunde beräkningstiden som grafikkortet krävde för en frame sparas ner. För att sedan påbörja datainsamlingen byggdes 28 olika 'development builds'. Detta gjordes för sju olika upplösningar med ett bildförhållande om 16:9 på eldsimuleringen samt fyra olika mängder för Jacobi-iterationerna per upplösning. Separata tester för varje konfiguration genomfördes därefter. Mätningarna gjordes i 'development builds' för att undvika påverkan från Unity-editorn. De byggda projekten var markerade som 'development builds', alltså utvecklar-versioner, för att tillåta Recorder-verktyget att samla in data.

Från att processen startades tills att 10 000 frames hade slutförts samlades vid varje frame tiden det tog för grafikkortet att slutföra simuleringen. Varje tid adderades sedan ihop med den sammanlagda tiden från föregående frames och vid simuleringens slut dividerades resultatet med 10 000 vilket var antalet frames. Detta gav ett medelvärde som representerade den genomsnittliga tiden det tar för grafikkortet att göra en simulering, givet en viss upplösning och mängd Jacobi-iterationer.

Anledningen till att repetera varje mätning 10 000 gånger är för att få ett pålitligt medelvärde som är mindre påverkat av andra faktorer, till exempel någon annan process på datorn påverkar prestandan eller andra möjliga Unity-processer som stör mätningen.

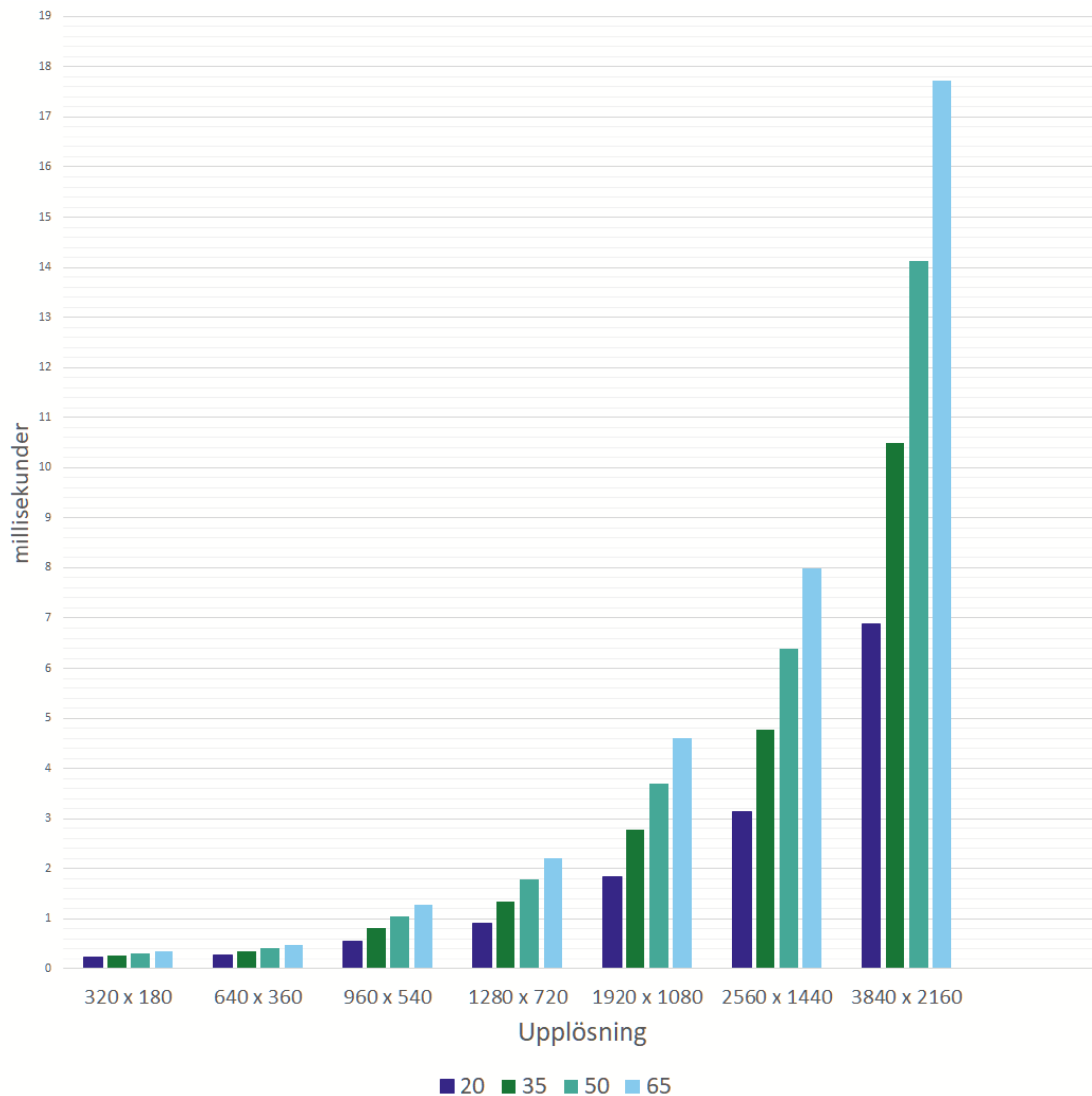
4.2.3 Framställning av grafer

När experimentet genomförts för alla upplösningar och Jacobi-iterationer kunde datan sammanställas till en serie grafer. Detta utfördes genom att skriva ner upplösningen eller mängden pixlar på x-axeln, tiden i millisekunder på y-axeln och för att representera Jacobi-iterationerna skapades ytterligare staplar av varierande färg för varje upplösning för att representera de olika mängden iterationer.

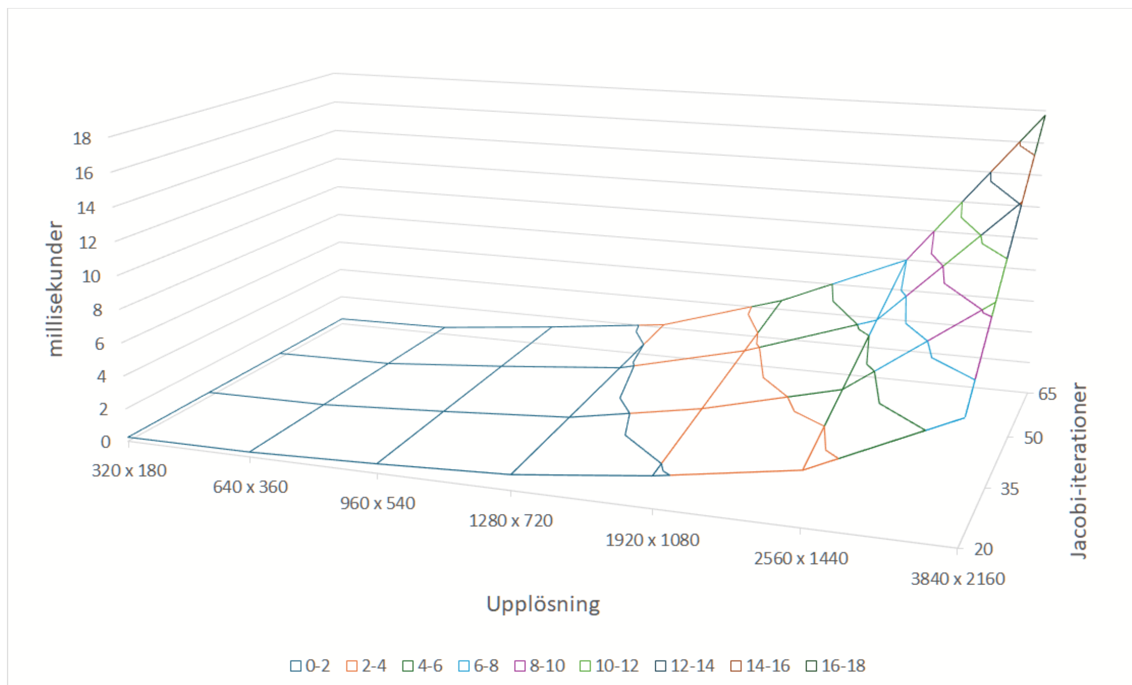
4.3 Resultat

Upplösning	Jacobi 20, uppmätt tid	Jacobi 35, uppmätt tid	Jacobi 50, uppmätt tid	Jacobi 65, uppmätt tid
320 x 180	0,24 ms	0,28 ms	0,30 ms	0,36 ms
640 x 360	0,29 ms	0,35 ms	0,42 ms	0,49 ms
960 x 540	0,56 ms	0,81 ms	1,05 ms	1,29 ms
1280 x 720	0,92 ms	1,34 ms	1,78 ms	2,20 ms
1920 x 1080	1,84 ms	2,76 ms	3,69 ms	4,60 ms
2560 x 1440	3,15 ms	4,77 ms	6,39 ms	7,99 ms
3840 x 2160	6,88 ms	10,49 ms	14,13 ms	17,72 ms

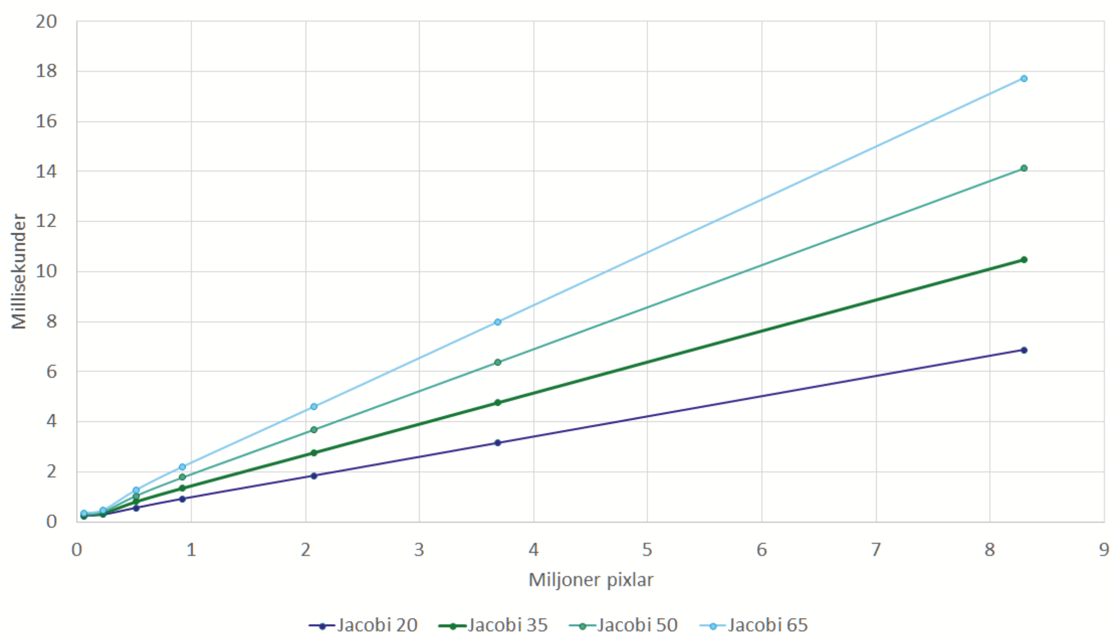
Tabell 2: Tabell över samtliga uppmätta medelvärden för 10 000 samplingar per upplösning samt per antal Jacobi-iterationer som samlades in. Värdena presenteras i storleken millisekunder.



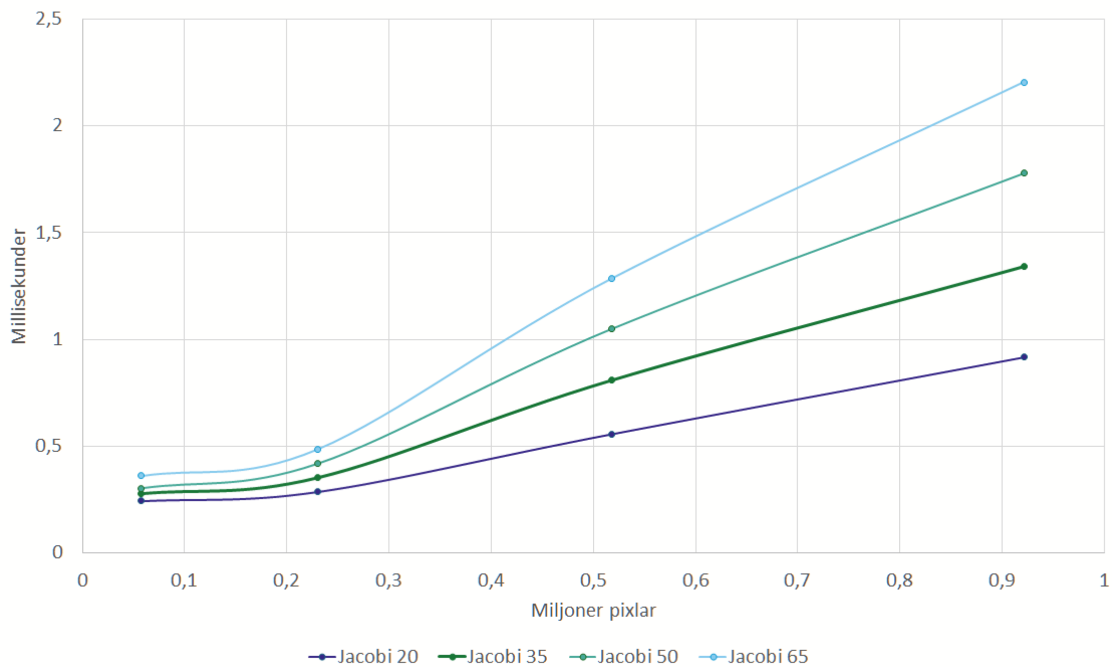
Figur 19: Samtliga mätresultat i stapeldiagram. Diagrammet består av sju olika upplösningar för vilka beräkningstiden uppmätts över 10 000 frames och den genomsnittliga beräkningstiden har därefter räknats ut och noteras i y-axeln i millisekunder. För varje upplösning visas också fyra olika staplar per upplösning där färgen på staplarna motsvarar mängden Jacobi-iterationer. Värt att notera är att storleken för upplösningarna inte följer något fast intervall.



Figur 20: Ytdiagram över samtliga mätresultat.



Figur 21: Samtliga mätresultat med utjämnad linje. Ett linjediagram med punkter för mätdata presenteras här för att bättre ge en uppfattning om storleksskillnaden i antalet pixlar mellan de upplösningar som undersöktes.



Figur 22: Diagram över de fyra lägsta upplösningarna, för en bättre översikt över dessa.

4.4 Analys

Som framgår vid inspektion av figurerna i avsnitt 4.3 har upplösningen en stor påverkan på den uppmätta tiden för alla olika Jacobi-iterationer. Omvänt gäller också där antalet Jacobi-iterationer har en stor påverkan på uppmätt tid för alla upplösningar. Samtliga mätresultat kan också ses i tabell 2.

Värt att notera är att det inte är något fast intervall mellan de olika upplösningarna avseende antalet pixlar. Därför är det svårt att säga något konkret angående vilket förhållande som gäller mellan simuleringens upplösning och den uppmätta tiden. I figur 21 och 22 antyds en övergripande trend som påminner om ett linjärt förhållande med undantag för de lägre upplösningarna. Detta säger dock inget om det egentliga förhållandet vilket skulle kräva mer noggranna mätmetoder än de metoder som används i detta arbete för att ge ett pålitligt resultat. Vi kan alltså inte redovisa en funktion för nämnda förhållanden. Dock kan en approximation göras genom att utföra en multipel linjär regressionsanalys på den data som insamlats.

Montgomery, Peck och Vining (2012, Kap: 3.1 MULTIPLE REGRESSION MODELS) beskriver hur multipel linjär regressionsanalys kan användas som en metod för att uppskatta en linjär funktion som beskriver förhållandet mellan två eller fler oberoende variabler och en beroende variabel när en sådan funktion inte är känd. Montgomery, Peck och Vining (2012, Kap: 3.3.1 Test for Significance of Regression) nämner också hur man ur en multipel linjär regression kan ta ut värdet R^2 vilket beskriver hur väl de faktiska variablerna stämmer överens med den uppskattade linjära funktionen. R^2 visar mellan 0 och 1 hur nära de faktiska variablerna stämmer med den uppskattade funktionen där ett R^2-värde närmare 0 antyder att funktionen stämmer mindre och ett R^2-värde närmare 1 antyder att funktionen stämmer mer.

Microsoft Excel tillåter oss att utföra en multipel linjär regressionsanalys av den data som insamlats. Regressionsanalysen görs på båda de oberoende variablerna, Jacobi-iterationer och antal pixlar, samt den beroende variabeln uppmätt tid, från experimentet. Analysen ger oss bland annat värdet för R^2.

Figur 23: Residualdiagram som visar residualerna för Jacobi-iterationer, alltså avståndet mellan uppmätt data och den uppskattade funktionen.

Figur 24: Residualdiagram som visar residualerna för upplösningar, alltså avståndet mellan uppmätt data och den uppskattade funktionen.

I figur 23 och figur 24 kan man utläsa residualerna från regressionsanalysen vilket enligt Montgomery, Peck och Vining (2012, Kap: 4.2.1 Definition of Residuals) visar på den konkreta avvikelsen för varje uppmätt tid från den genom regressionsanalysen uppskattade linjära funktionen. Detta är ett sätt att studera mer detaljerat vad det är som leder till det R^2-värde som analysen visar. R^2-värdet är ett resultat av residualerna.

Regressionsstatistik	
Multipel-R	0,95
R-kvadrat	0,91
Justerad R-kvadrat	0,90
Standardfel	1,41
Observationer	28

Figur 25: Regressionsstatistik från Microsoft Excel vid utförd analys av data från experimentet.

$$R^2 = 1 - \frac{SS_{Res}}{SS_T}$$

Ekvation 28: SS_{Res} är ett mått på variabiliteten där hänsyn tagits till oberoende variabler. SS_T är variabiliteten utan att hänsyn tagits till oberoende variabler. - Montgomery, Peck och Vining (2012, Kap: 2.6 COEFFICIENT OF DETERMINATION).

R^2-värdet som redovisas i figur 25 är ett resultat av residualerna och kan förklaras med hjälp av ekvation 28. R^2-värdet från vår analys är 0,91. Eftersom detta värde ligger nära 1 kan man tolka detta som att R^2 i detta fall antyder att den uppskattade linjära funktionen och uppmätt data från experimentet stämmer överens ganska väl. Detta kan tolkas som att förhållandet mellan den beroende variabeln uppmätt tid och de två oberoende variablerna Jacobi-iterationer och antal pixlar påminner om ett linjärt förhållande vilket stämmer överens med den visuella tolkningen som gjorts av figur 21 och figur 22.

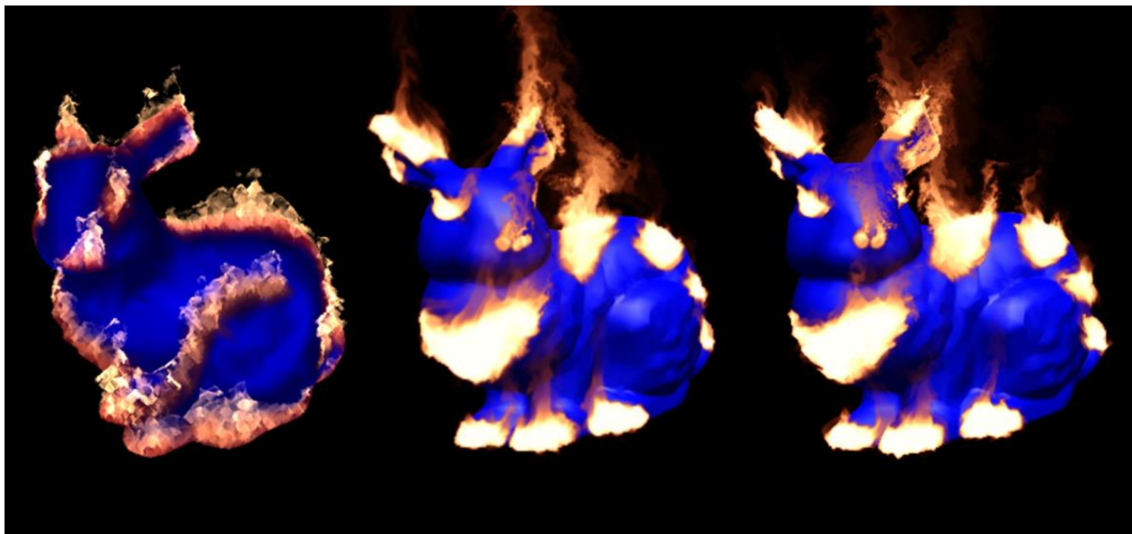
Alltså kan slutsatsen dras att förhållandet mellan uppmätt tid och antal pixlar i experimentet samt förhållandet mellan uppmätt tid och antal Jacobi-iterationer i experimentet påminner om ett linjärt förhållande baserat på visuell analys samt en multipel linjär regressionsanalys. Residualerna antyder att de uppmätta tiderna avviker något från den uppskattade linjära funktionen och därför kan inget definitivt sägas om förhållandena. Dock ger detta en indikation på förhållandenas karaktär vilket påminner om ett linjärt förhållande.

Vid granskning av figur 19, figur 20 och figur 21 syns att alla uppmätta tider förutom en tid är lägre än 16,7ms vilket är tiden för en bildruta vid 60 bilder per sekund. Det värde som överstiger 16,7ms är också det mätvärde där upplösningen samt mängden Jacobi-iterationer var de största som ingick i experimentet. Det skall påpekas att för ett datorspel som skall uppnå minst 60 bilder per sekund är det lämpligt att det finns tid över för andra beräkningar än endast eldsimuleringen då detta sannolikt inte skulle vara det enda innehållet i spelet. Det kan mot bakgrund av detta vara av intresse att konstatera att alla mätvärden i experimentet förutom de tre högsta lämnar minst hälften av beräkningsbudgeten vid 60 bilder per sekund över för andra beräkningar på grafikkortet. Sannolikt är det inte rimligt att lägga hälften av beräkningstiden på en enskild visuell effekt. Detta beror dock på vilken typ av spel som effekten skall användas i. Mer intressant är att elva av konfigurationerna ligger under 1 millisekund. Detta kan ses som en mer realistisk fördelning av beräkningsbudgeten. 1 millisekund är ca 1/16 av tiden för 60 bilder per sekund.

Vid implementering av eldsimuleringen som presenteras i detta arbete i ett datorspel kan följande antagande göras. Vid implementeringen måste hänsyn tas till den budget gällande

beräkningstid på grafikkortet som måste hållas för att uppnå önskad bilduppdateringsfrekvens. Alltså är det rimligt att vid utveckling av ett datorspel med många grafiktunga beräkningar utöver eldsimuleringen konfigurera eldsimuleringen med några av de lägre värdena för upplösning samt mängd Jacobi-iterationer. Däremot kan ett datorspel med enklare beräkningar för övrig grafik tillåtas en simuleringskonfiguration med högre värden för upplösning samt antal Jacobi-iterationer.

En visuell bedömning gjordes under experimentet avseende den visuella kvaliteten på eldsimuleringen, och till vilken grad den överensstämde med effekten som presenteras i Guay, Colin och Egli(2011). Enligt bedömningen var de konfigurationer som mest liknade effekten hos förlagan samt hade högst visuell kvalitet, de med upplösningarna 960 x 540 och 1280 x 720 samt 50 Jacobi-iterationer. En visuell representation av dessa konfigurationer samt simuleringen i Guay, Colin och Egli(2011) kan ses i figur 26. Mätvärdena för dessa ligger båda under 2 millisekunder.



Figur 26: Från vänster till höger: bild ur simulering från Guay, Colin och Egli(2011), bild ur simulering motsvarande experimentet vid 960x540 och 50 Jacobi-iterationer, bild ur simulering motsvarande experimentet vid 1280x720 och 50 Jacobi-iterationer.

Före experimentets utförande gjordes ett antagande kring att eldsimuleringen skulle prestera relativt väl baserat på de goda resultat som presenteras Guay, Colin och Egli(2011) samt den beskrivning av simuleringsmetodens effektivitet som Harris(2007) nämner. Vid granskning av insamlad data från experimentet förefaller detta antagande stämma relativt väl, åtminstone för den del av konfigurationerna som har ett uppmätt medelvärde på under 2 millisekunder. Det är svårt att sätta en konkret gräns för vad som anses vara prestandaeffektivt men sannolikt finns det många tänkbara applikationer för en eldsimulering som inte överstiger ca 2 millisekunder.

Sammanfattningsvis kan mot bakgrund av frågeställningen konstateras att prestandapåverkan från eldsimuleringen är avsevärd. Dock visar flera uppmätta resultat att simuleringen kan konfigureras för att ge mycket tid över för övriga grafikberäkningar vid ett mål om 60 bilder per sekund. Resultaten visar endast hur prestandan ser ut för de konfigurationer som vid visuell bedömning ansågs ge en önskad kvalitet på effekten och inget linjärt förhållande kunde konstateras avseende relationen mellan upplösning och uppmätt tid eller relationen mellan antal Jacobi-iterationer och uppmätt tid. Dock antyder diagrammen att förhållandet visuellt påminner om ett linjärt sådant. Detta stämmer också med resultaten från regressionsanalysen. Detta ger möjlighet att med viss säkerhet uppskatta

ungefär hur olika konfigurationer skulle prestera. Det går alltså att säga att upplösning samt mängd Jacobi-iterationer är bra verktyg för att optimera simuleringen förutsatt att hänsyn tas till önskad visuell kvalitet.

5 Sammanfattning och diskussion

5.1 Sammanfattning

Simulering av eld med målet att skapa en trovärdig visuell effekt kan uppnås genom tekniker för vätskesimulering (Guay, Colin och Egli 2011). Harris(2007) och Guay, Colin och Egli(2011) presenterar båda metoder för hur denna typ av vätskesimulering kan implementeras på ett effektivt sätt med hjälp av parallellisering på grafikkortet där den senare även redogör för en nyskapande metod för att rendera elden på ett trovärdigt sätt i två dimensioner där illusionen av en tredje dimension uppnås genom att delvis göra ett undantag för villkoret för icke-komprimerbarhet i simuleringen.

I detta arbete föreslås en upprepning av dessa metoder i Unity för att kunna utvärdera dess lämplighet för praktiskt användande i datorspel. Omfattningen på arbetet begränsar utvärderingen till att enbart undersöka lösningens prestanda för att se vilken prestandapåverkan som kan uppmätas och ett experiment utformades för att undersöka beräkningstiden för ett simuleringssteg i simuleringen på grafikkortet.

En implementation av nämnda simuleringsmetoder gjordes med Hawkins(2017) källkod baserad på Harris(2007) som grund för implementationen där målet var att efterlikna den effekt som presenteras i Guay, Colin och Egli(2011) rent visuellt. Lösningen modifierades i huvudsak gällande rastrering av objekt i scenen som bränsle för elden samt gällande en uppskalning av tryckgradienten för att som beskrivs i Guay, Colin och Egli(2011) skapa tryckvågor i velocitetsfältet som ger illusionen av rörelse i djupled.

För att mäta prestandan bestämdes ett antal konfigurationer för simuleringen avseende upplösning och antal Jacobi-iterationer. För varje konfiguration kördes simuleringen över 10 000 uppdateringscykler varpå den genomsnittliga beräkningstiden för cyklerna beräknades med hjälp av Unitys Recorder-API. Resultaten sammanställdes i en serie diagram och en analys gjordes med hjälp av dessa.

Resultaten visade att prestandapåverkan från eldsimuleringen var avsevärd men att det går att styra prestandapåverkan genom att variera upplösning samt antal Jacobi-iterationer med hänsyn till önskad visuell kvalitet på effekten. Det uppmärksammades att resultaten antyder en trend som visuellt påminner om ett linjärt förhållande mellan uppmätt tid och upplösning samt uppmätt tid och antal Jacobi-iterationer men det konstaterades vid närmare undersökning att förhållandena i själva verket inte är linjära. Det noterades också att majoriteten av de testade konfigurationerna gav en uppmätt tid som var lägre än högst 16,7ms och att flera resultat var ännu lägre.

5.2 Diskussion

De främsta styrkorna med den metod för eldsimulering som beskrivits är dess relativt goda prestanda samt den övertygande visuella effekten. Det kan argumenteras för att lösningen som redovisats egentligen är förhållandevis tungdriven då prestandapåverkan är högst påtaglig. Men i ljuset av att både Stam(2003) och Green och Horvath(2012) påpekar att vätskesimulering i stort är ett prestandatungt område samt att implementationen som redovisats i detta arbete bygger på Harris(2007) argumenterbart effektiva GPU-implementation kan man rimligtvis påstå att metoden som presenterats bör ses som förhållandevis effektiv. Detta kan styrkas med det faktum att prestandan går att påverka

genom vilken konfiguration av simuleringen som väljs. Den förhållandevis realistiska och övertygande visuella eldeffekten påpekas också av Guay, Colin och Egli(2011) för deras egen simulering. Eldsimuleringen i detta arbete ger ett visuellt resultat som är mycket likt deras.

Mot bakgrund av resultaten gällande prestanda och visuell kvalitet kan det argumenteras för att nämnda eldsimulering åtminstone gällande vad som undersökts i detta arbete kan anses vara lämplig för användning i utveckling av datorspel. Undersökningen är dock begränsad i omfattning och vidare undersökningar skulle vara lämpliga för en mer komplett bedömning av lämpligheten. Ett exempel på vad som hade varit en rimlig undersökning är att titta på eventuella visuella artefakter som kan uppstå i olika typer av 3D-miljöer samt vid olika kamerarörelser. I detta arbete valdes en mycket enkel 3D-miljö med enstaka objekt samt en orörlig kamera. Guay, Colin och Egli(2011) påpekar hur det i deras implementation uppstår visuella artefakter vid snabba kamerarörelser eller vid snabb förflyttning av brinnande objekt och att en möjlig lösning på detta problem kan vara att översätta kamerans transformationer via en matris till screen-space för att flytta den simulerade elden i enlighet med kamerans rörelser. Detta är något som hade varit applicerbart även i detta arbete då liknande artefakter förekommer även i vår implementation och detta är rimligtvis något som bör tas hänsyn till vid användning av eldsimuleringen i ett datorspel med mycket rörelse. Med andra ord är metodens känslighet för rörelse begränsande för dess användningsbarhet.

Under arbetets gång dök en rad frågor upp kring hur eldsimuleringen skulle bete sig i olika sammanhang. Exempel på dessa är vad som händer vid simulering av eld på flera objekt som överlappar varandra i djupled eller hur simuleringen hanterar en stor platt yta som brinner, till exempel ett gräsfält liknande de i Far Cry 2 (2006). Dessa frågor hade varit intressanta att undersöka och bör lämpligtvis tas hänsyn till vid användning av eldeffekten beroende på sammanhanget i vilket effekten skall användas.

I Harris(2007) nämns metoder för så kallad 'vorticity confinement' vilket syftar till att öka turbulensen i simuleringen. Detta kan tänkas höja kvaliteten på elden som simuleras på ett liknande sätt som den metod vi använt gällande uppskalning av tryckgradienten. Det beslutades att inte implementera någon 'vorticity confinement' i detta arbete på grund av begränsad tid men det är en aspekt som verkar rimlig att undersöka vidare för att se ifall detta ger en ännu mer övertygande visuell effekt.

Under arbetets gång har bedömningen gjorts att den eldeffekt som uppnås med den redovisade metoden mycket riktigt tycks ge en mer realistisk eld än vad som vanligen syns vid effekter baserade på enkla 'sprite sheet'-animationer eller partikelsystem. Dock skulle detta behöva verifieras genom till exempel en enkätundersökning där en mängd testpersoner får göra en subjektiv bedömning. Detta skulle då kunna ge en uppfattning om effektens värde sett till visuell kvalitet i förhållande till prestandakostnad och kontexten för en applikation där någon eldeffekt önskas. I nuläget är det svårt att ge en konkret bedömning för huruvida eldsimuleringen som redovisas i detta arbete generellt kan sägas vara bättre än till exempel en partikelbaserad effekt men detta är inte heller tanken bakom arbetet utan arbetet är snarare att betrakta som en möjlig referens som del i en helhetsbedömning kring effektens lämplighet för praktisk användning i datorspel.

5.3 Samhälleliga och etiska aspekter

Simulering av fluider är vanligt förekommande i sammanhang där verkliga vätskeflöden behöver simuleras på ett naturtroget sätt och Navier-Stokes kan som Kellomäki(2017) nämner ses som en central del i detta område. Ett exempel på detta som nämns i Miao et al(2023) är hur LiDAR

användes tillsammans med strömningsmekanik för att simulera olika översvämningsscenarier i Nederländerna. Implementationen i detta arbete skiljer sig dock från metoder som används inom områden där fysikalisk noggrannhet eftersträvas och de kritiska fysiska beräkningarna ur vår implementation kan därför inte användas för att beräkna fenomen där ett fysiskt korrekt resultat är viktigt(Stam 2003). Ett exempel på en fysikaliskt icke-korrekt beräkning är hur flera värden manipuleras för att ge en tredimensionell effekt till elden(Guay, Colin och Egli 2011).

Den redovisade metoden har dock flera användningsområden inom digital media som datorspel eller liknande där en illusion av eld önskas med, som Kellomäki(2017) påpekar, mindre prestandapåverkan än mer fysikaliskt korrekta metoder. Specifika fördelar med detta arbetets implementation är en mindre prestandapåverkan än vad som hade krävs för en tredimensionell simulering, då endast en illusion av tre dimensioner ges och den totala beräkningskostnaden för att simulera tre dimensioner kan därför undvikas (Guay, Colin och Egli 2011). Detta leder till att implementeringen som nämns i Guay, Colin och Egli(2011) inte är en fysiskt korrekt simulering då olika värden är överdrivna för att skapa olika visuella effekter.

5.4 Framtida arbete

I detta arbete har vi presenterat vilken slags prestandapåverkan en Navier-Stokes baserad eld implementation i Unity kan ha. Framtida utveckling involverar bland annat utforskning kring hur förflyttning av eldeffekten kan ske utan att introducera visuella artefakter. Ett exempel på detta är som nämnts tidigare i diskussionen en lösning där kamerans transformationer översätts till eldsimuleringen genom någon matrisberäkning. Utan någon vidare utveckling av detta uppstår flera grafiska artefakter vid förflyttning av objektet som simuleringen utförs på eller vid förflyttning av kameran.

Ett annat område som detta arbetet inte undersökte med noggrannhet är estetiska aspekter. Då ett estetiskt perspektiv krävt en annan form av datainsamling ansågs det vara utanför arbetets omfattning. Det är därför lämpligt att framtida forskning antar ett mer estetiskt perspektiv och analyserar påverkan av denna implementering på den visuella presentationen i någon lämplig applikation där eldeffekten skall implementeras. Detta område skulle även inkludera en analys av potentiella grafiska artefakter eller möjliga problem som kan uppstå. Detta kräver utförliga tester för effekterna av olika miljöer och spelsammanhang som detta arbete inte innefattar.

Referenser

Claypool, K.T. Claypool, M. (2007) *On frame rate and player performance in first person shooter games*, *Multimedia Systems*, 13(1), ss. 3–17. doi:10.1007/s00530-007-0081-1.

Eliasson, A. (2019). *Kvantitativ metod från början*. (4:e uppl.) Lund: Studentlitteratur.

Epic Games (u.å.) *Fluid Simulation Overview, Epic Developer Community. Epic Games*. https://dev.epicgames.com/documentation/en-us/unreal-engine/fluid-simulation-in-unreal-engine---overview?application_version=5.0 [2024-05-20]

Far Cry 2 (2006). Ubisoft.

Gibiansky, A (2011) *Fluid Dynamics: The Navier-Stokes Equations* https://andrew.gibiansky.com/blog/physics/fluid-dynamics-the-navier-stokes-equations/ [2024-03-05]

Green, S. Horvath, C. (2012). *Flame On: Real-Time Fire Simulation for Video Games* [PowerPoint-presentation]. Nvidia GTC. https://on-demand.gputechconf.com/gtc/2012/presentations/S0102-Flame-on-RT-Fire-Simulation-for-Video-Games.pdf [2024-05-20]

Guay, M. Colin, F. och Egli, R. (2011) *Screen space animation of fire*, *SIGGRAPH Asia 2011 Sketches*, ss. 1–2. doi:10.1145/2077378.2077391.

Gunadi, S.I. Yugopuspito, P. (2018) *Real-Time GPU-based SPH Fluid Simulation Using Vulkan and OpenGL Compute Shaders*, *2018 4th International Conference on Science and Technology (ICST),* ss. 1–6. doi:10.1109/ICSTC.2018.8528699.

Harris, J M (2007) *Chapter 38. Fast Fluid Dynamics Simulation on the GPU, GPU Gems https://developer.nvidia.com/gpugems/gpugems/preface*

Hawkins, J (2017) *GPU-GEMS-2D-Fluid-Simulation, https://github.com/Scrawk/GPU-GEMS-2D-Fluid-Simulation.git* [2024-03-13]

Kellomäki, T. (2017) '*Fast Water Simulation Methods for Games*', *Computers in Entertainment*, 16(1), pp. 1–14. doi:10.1145/2700533.

Miao, J.L. *et al.* (2023) '*Interactive Simulation of Realistic Fluid Movement Based on SPH Method*', *International Journal of Multiphysics*, 17(2), pp. 203–216. doi:10.21152/1750-9548.17.2.203.

Montgomery, D.C, Peck, E.A. och Vining, G.G. (2012) *Introduction to linear regression analysis. 5. ed. Wiley (Wiley series in probability and statistics: 821).* https://search.ebscohost.com/login.aspx?direct=true&db=cat09042a&AN=his.oai.his.se.21799&lang=sv&site=eds-live [2024-05-22]

Stam, J. (1999) 'Stable fluids', *Proceedings of the 26th Annual Conference: Computer Graphics & Interactive Techniques*, pp. 121–128. doi:10.1145/311535.311548.

Stam, J. (2003) *Real-Time Fluid Dynamics for Games, Proceedings of the game developer conference* Vol. 18, s. 25. https://www.dgp.toronto.edu/public_user/stam/reality/Research/pdf/GDC03.pdf

TechPowerUp (u.å.) NVIDIA GeForce 9800 GT Specs. https://www.techpowerup.com/gpu-specs/geforce-9800-gt.c635 [2024-02-15]

TechPowerUp (u.å.) NVIDIA GeForce RTX 3070 Specs. https://www.techpowerup.com/gpu-specs/geforce-rtx-3070.c3674 [2024-02-15]

Toftedahl, M. och Engström, H. (2019) '*A Taxonomy of Game Engines and the Tools that Drive the Industry', Game Hub Scandinavia DiGRA '19 - Proceedings of the 2019 DiGRA International Conference Digital Games Research Association (DiGRA)* [Preprint]. https://search.ebscohost.com/login.aspx?direct=true&db=edsswe&AN=edsswe.oai.DiVA.org.his.17706&lang=sv&site=eds-live [2024-05-19]

Unity Technologies (2020) Challenge: Creating Fire with Particle Systems. https://learn.unity.com/tutorial/challenge-creating-fire-with-particle-systems?projectId=5f078cfdedbc2a3231d47753# [2024-03-12]

Unity Technologies (2024) GPU Usage Profiler Module https://docs.unity3d.com/2022.3/Documentation/Manual/ProfilerGPU.html [2024-02-15]

Unity Technologies (2024) Scripting API: Recorder. https://docs.unity3d.com/ScriptReference/Profiling.Recorder.html [2024-03-11]

Unity Technologies (2024) Scripting API: CustomSampler. https://docs.unity3d.com/ScriptReference/Profiling.CustomSampler.html [2024-03-11]

Unity Technologies (2024) Long Term Support, Unity. https://unity.com/releases/editor/qa/lts-releases [2024-03-11]

Unity Technologies (2024) Scripting API: RenderTexture. https://docs.unity3d.com/ScriptReference/RenderTexture.html [2024-03-11]