Examensarbete

HÖGSKOLAN I SKÖVDE

JUST-IN-TIME-BASERAD LATENSREDUKTION I DATASPEL

Implementation och utvärdering av en ny teknik för att minska latens vid dubbelbuffrad vertikal synkronisering

JUST-IN-TIME-BASED LATENCY REDUCTION IN COMPUTER GAMES

Implementation and evaluation of a novel technique for decreasing latency when using double buffered vertical synchronization

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2022

Olle Axelsson

Handledare: Mikael Thieme
Examinator: Sanny Syberfeldt

Sammanfattning

Det är först nyligen som den lokala latensperioden hos datorsystem börjat undersökas och optimerats i dataspel och nya hårdvaru- och mjukvarutekniker har utvecklats för att minska latensen i dem. I detta arbete läggs en ny sådan teknik fram, inspirerad av Nvidia Reflex fast optimerad för dubbelbuffrad vertikal synkronisering. Tekniken implementerades i en egen spelmotor som sedan användes för att skapa en prestandatest-applikation. Med hjälp av denna applikation utfördes ett experiment för att utvärdera teknikens praktikalitet och prestanda. Resultatet från experimentet tyder på att tekniken kan göra god nytta för att minska latensen i vertikalt synkroniserade spel, men kräver för tillfället att implementationen optimeras utefter spelets prestanda. Framtida forskning bör fokusera på att öka generaliserbarheten av tekniken genom att exempelvis utnyttja maskinlärning. Dessutom önskas bättre stöd från grafikdrivrutiner och operativsystem att förse applikationer med de verktyg som behövs för att göra en solid implementation av denna teknik.

Nyckelord: dataspel, latens, just-in-time, spelmotor

Innehållsförteckning

1 Introduktion

Den lokala latensperioden hos ett datorsystem är något som knappt undersökts och optimerats i dataspel förrän väldigt nyligen. Det finns gott om vetenskapligt bevis för att latensperiod har en stor påverkan på spelarprestation. I och med det har nya tekniker utvecklats för att minska systemets latensperiod. En av dessa tekniker är Nvidia Reflex som kan minska latensperioden genom att dynamiskt fördröja när spelets uppdatering sker och i sin tur när inmatning hämtas, vilket gör att inmatning hämtas närmre det att bildrutan visas på skärmen. Dock fungerar inte denna latensreduceringsteknik tillsammans med dubbelbuffrad vertikal synkronisering.

Målet för detta arbete var att utveckla och implementera en liknande form av latensreduceringsteknik optimerad för att minska latens vid dubbelbuffrad vertikal synkronisering. För att göra detta behövde det göras en prognos av den framtida beräkningstiden för varje bildruta med god precision. Implementationen utvärderades därefter genom ett experiment där latens och prestanda mättes vid olika belastningar och med olika parametervärden i prognosberäkningen.

Verktygen som krävdes för att uppnå denna typ av latensreducering samlades till ett fristående bibliotek vid namn KaJit. Biblioteket innehåller verktyg för att samla in mätningar av beräkningstider hos CPU och GPU, samla in data om när det vertikala blankningsintervallet sker, göra prognoser av framtida beräkningstid och med hjälp av detta fördröja när uppdateringen sker för att reducera latens. Biblioteket implementerades därefter i den egenskrivna spelmotorn Kamera Engine, som i sin tur användes för att konstruera testapplikationen som användes under experimentet. KaJit är dock designat för att vara oberoende av spelmotor eller grafik-API och kan därför implementeras i andra program eller spelmotorer.

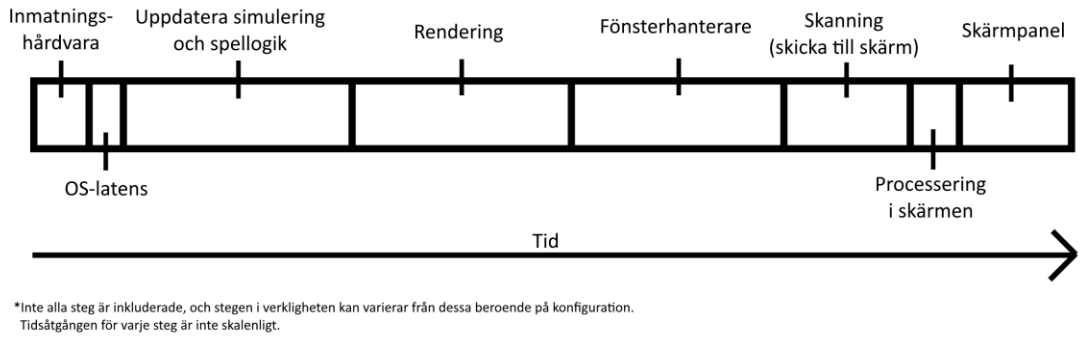
2 Bakgrund

Det finns ett flertal koncept som är viktiga att veta om för att förstå problemet som detta arbete behandlar. I detta kapitel kommer dessa koncept gås genom och deras relation till latensperiod förklaras.

2.1 Latens i spel och spelmotorer

Ordet ”latensperiod” definieras som ”tid när en process ännu inte är märkbar” (Svenska Akademien 2015). Inom datorer används ordet ”latensperiod” eller ”latens” för att beskriva tidsfördröjningen från att en inmatning sker till dess att konsekvensen eller resultatet av inmatningen kan observeras (PCMag u.å.). Det kan exempelvis referera till latensen från enskilda hårdvaru- eller mjukvarukomponenter eller den sammanlagda latensen från alla komponenter i ett datasystem.

I ett dataspel finns flera komponenter som kan medföra latens, inklusive inmatningshårdvara, simuleringar och beräkningar i spelet, rendering, att överföra bildrutan från grafikkortet till skärmen, bearbetning i skärmen, skärmpanelslatens, drivrutiner och olika delsteg i operativsystemet (Schneider 2020; Žilys 2020). Sammanlagt utgör dessa systemets latens, se figur 1. I sammanhanget av spel kallas systemets latens ofta för ”input lag”, men Schneider (2020) kritiserar denna term eftersom den även används för att referera till latensen hos enskilda komponenter.



Figur 1 En illustration som visar hur olika komponenter i ett dataspel kan bidra till latens.

Ett av stadierna som Schneider (2020) säger ofta har störst påverkan på systemets latens är den interna latensen i spelet självt. Det är tiden från det att spelet läser av användarens inmatning till det att bildrutan är färdigrenderad och redo att visas på skärmen (ibid.). Spelets interna latens kan påverkas av ett flertal olika faktorer. Att använda vertikal synkronisering är ett exempel på en faktor som kan öka latensen (Digital Foundry 2018). Ett annat exempel är att spel vanligtvis använder sig av en så kallad renderingskö för att låta processorn beräkna flera tidssteg i förväg för att minska risken för lagg, på bekostnad av en högre latens (Schneider 2020; Žilys 2020).

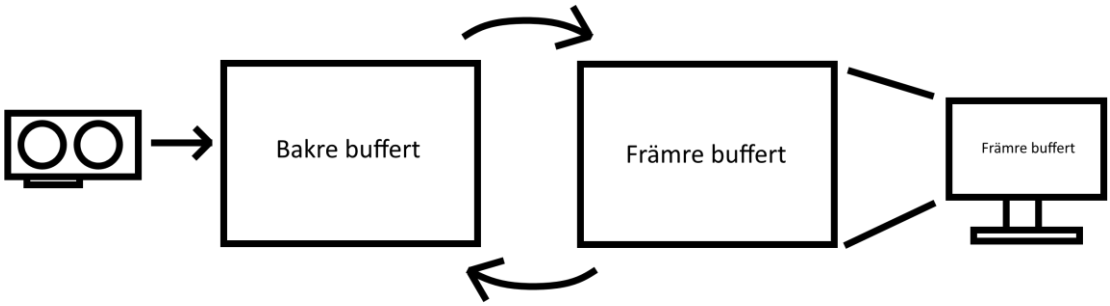
2.2 Rendering

2.2.1 Dubbelbuffrad rendering

En skärmbuffert är ett område i grafikminnet som representerar en bild som grafikkortet kan rendera till och som kan visas på skärmen. I ett program som är i exklusivt helskärmsläge motsvarar skärmbufferten exakt den bild som visas på skärmen. I ett program som är i fönsterläge används skärmbufferten av operativsystemets fönsterhanterare för att visa programmet i ett fönster (Oracle u.å.).

Om bara en skärmbuffert används visas skärmbufferten hela tiden, även när den renderas till. Det innebär att skärmbufferten inte garanterat har en färdigrenderad bild när den visas på skärmen (Oracle u.å.). I program där innehållet på skärmen sällan ändras eller bara ändras lite syns det knappt, men så fort mer komplex rendering med flera lager utförs blir artefakterna tydliga (Embedded Wizard u.å.).

Lösningen är att använda två skärmbuffertar, en ”främre” buffert som är en färdig bild som visas på skärmen och en ”bakre” buffert som är dold och kan renderas till utan att det visas på skärmen. När den bakre bufferten är färdigrenderad byter den bakre och främre bufferten plats, så den bakre bufferten blir den främre bufferten och vice versa[1], se figur 2. På så sätt visas enbart färdigrenderade bilder på bildskärmen. Denna teknik kallas för dubbelbuffrad rendering (Oracle u.å.).

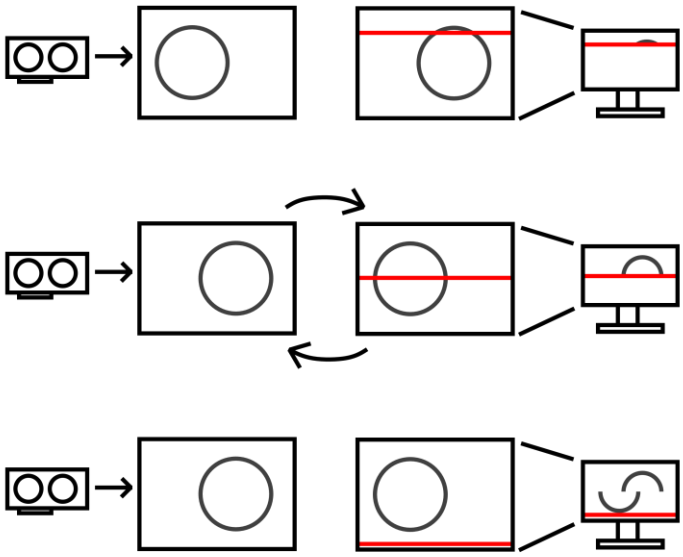


Figur 2 En illustration som visar dubbelbuffrad rendering.

2.2.2 Tearing och vertikal synkronisering

Grafikkortet skickar skärminnehållet rad för rad uppifrån och ned till skärmen i en process som kallas för skanning (The kernel development community u.å.). Efter att varje bild skannats ut är det en kort paus innan nästa bild skannas ut som kallas för det vertikala blankningsintervallet (ibid.). Om den bakre och främre skärmbufferten byter plats under skanningen uppstår det en vertikal skärning i bilden som visas på skärmen, där två olika bilder visas på olika områden av skärmen (ibid.), se figur 3. Denna artefakt kallas för tearing. För att undvika tearing måste buffertbytet ske under det vertikala blankningsintervallet (ibid.). Att synkronisera buffertbytet med det vertikala blankningsintervallet är en teknik som kallas för vertikal synkronisering eller vsync (Digital Foundry 2018).

[1] Notera att buffertarna inte fysiskt byter plats i minnet, utan enbart referenserna till buffertarna byter plats. I vissa implementationer av dubbelbuffrad rendering kopieras i stället innehållet från den bakre bufferten till den främre bufferten, men detta medför en prestandakostnad som är onödig om den kan undvikas (Oracle u.å.). Fönsterhanteraren i Windows 7 och senare har stöd för att använda referenser till skärmbuffertar vid sammansättning (Microsoft 2021a).



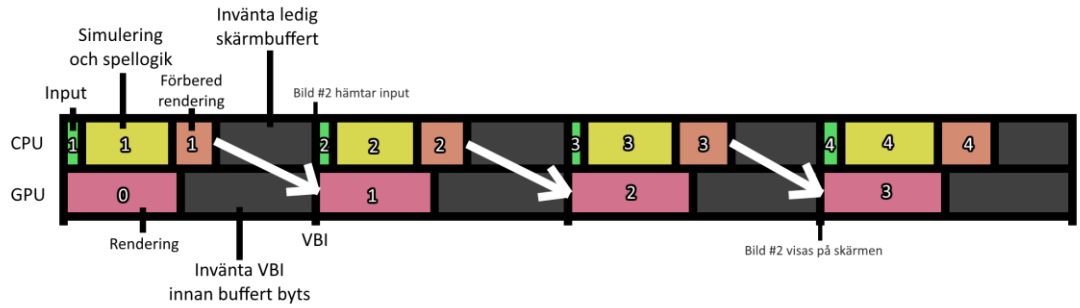
Figur 3 En illustration som visar hur tearing uppstår. Den röda linjen representerar raden som skannas till skärmen vid det ögonblicket. Ett buffertbyte sker mitt under skanning.

2.2.3 Dubbelbuffrad vertikal synkronisering

När dubbelbuffrad rendering används tillsammans med vertikal synkronisering väntar programmet på det vertikala blankningsintervallet när den bakre bufferten är färdigrenderad och därefter byts buffertarna. När endast två skärmbuffertar används går det inte att rendera ytterligare bildrutor medan den bakre bufferten väntar på det vertikala blankningsintervallet. Den främre skärmbufferten är upptagen med att skannas ut och den bakre skärmbufferten har redan en färdig bild som kan behöva visas på skärmen när som helst. Hade den målats om finns risk att det vertikala blankningsintervallet missas i onödan alternativt att en ej färdigritad bild måste visas. Därför innebär vertikal synkronisering med dubbelbuffring även att spelets uppdateringsfrekvens begränsas till en submultipel [2] av skärmens uppdateringsfrekvens (Digital Foundry 2018).

Denna väntan på att skärmbuffertarna ska bytas orsakar en ökning i latens av två anledningar. Den första anledningen är att nya färdigrenderade bildrutor inte kan visas omedelbart utan måste vänta till nästa vertikala synkroniseringsintervall för att visas (Digital Foundry 2018). Den andra ökningen kommer från den begränsade uppdateringsfrekvensen i spelet, som i egenskap av att alltid vara en submultipel av skärmens uppdateringsfrekvens aldrig kan vara högre än skärmens uppdateringsfrekvens. För att göra saker värre hinner beräkningarna ofta bli klara i god tid innan det vertikala synkroniseringsintervallet, vilket innebär att mycket tid därefter spenderas på att bara vänta in det vertikala synkroniseringsintervallet (Žilys 2020), se figur 4. Det gör att det i många fall går onödigt lång tid mellan att inmatning läses av till att bildrutan visas.

[2] En ”submultipel” är inversen av en multipel. Om a är en submultipel av b är $\frac{b}{a}$ ett heltal (Svenska Akademien 1997).



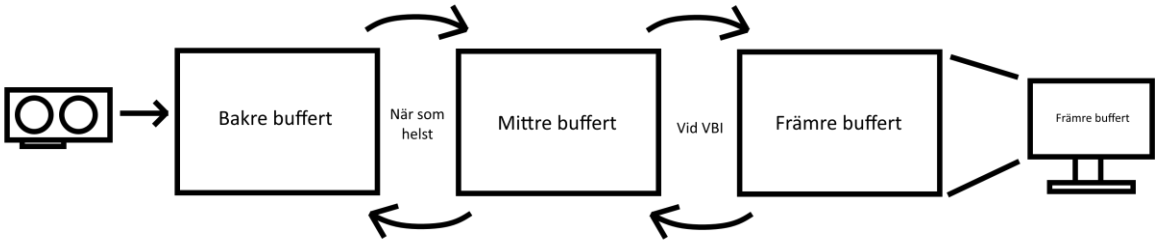
Figur 4 En illustration som visar hur väntan på det vertikala blankningsintervallet (”VBI” i bilden) kan leda till hög latens.

Fördelen med vertikal synkronisering vid dubbelbuffrad rendering är att det ger jämn rörelse så länge inte lagg uppstår, eftersom det då går lika lång tid i simuleringen mellan varje bildruta som visas på skärmen[3] (Digital Foundry 2018). Det förhindrar även att fler bildrutor renderas än vad bildskärmen kan visa, vilket är energieffektivt (Korpela 2014).

2.2.4 Trippelbuffrad vertikal synkronisering

För att undvika tearing utan att begränsa uppdateringsfrekvensen, och därmed undvika den latens som vänteperioderna skapar, går det att använda trippelbuffring. Då används ytterligare en dold skärmbuffert, den mittre skärmbufferten, för att koppla loss spelets uppdateringshastighet från bildskärmens uppdateringsfrekvens (ARM 2011).

Grafikkortet renderar precis som innan till den bakre bufferten. När en bildruta är färdigrenderad byter den bakre och mittre bufferten plats omedelbart[4], vilket gör det direkt går att börja arbeta på en ny bildruta i den bakre bufferten. Den främre och mittre bufferten däremot byter plats vid varje vertikalt synkroniseringsintervall om en ny bildruta finns i den mittre bufferten. Buffertbytet mellan den främre och mittre bufferten är alltså kopplat till det vertikala synkroniseringsintervallet men bytet mellan den bakre och mittre bufferten är inte det. Det innebär att spelet kan fortsätta arbeta på nya bildrutor oberoende av skärmens uppdateringsfrekvens (ARM 2011). Se figur 5.



Figur 5 En illustration som visar trippelbuffrad rendering.

Nackdelen med trippelbuffring är att det går miste om rörelsejämnhets- och energieffektivitetsaspekterna av dubbelbuffrad rendering, precis som dubbelbuffring utan

[3] Som Žilys (2020) beskriver är det viktigt att tidssteget beräknas på ett bra sätt för att uppnå så jämna tidssteg och därmed så jämn rörelse som möjligt.
[4] Den bakre och mittre bufferten kan byta plats även om bildrutan som ligger i den mittre bufferten aldrig målats ut på skärmen (ARM 2011).

vertikal synkronisering (Digital Foundry 2018). Trippelbuffring kräver dessutom ytterligare en skärmbuffert till i grafikminnet (ARM 2011).

2.3 Pipelining i spel

Eftersom dagens moderna datorer har flera kärnor och använder ett separat grafikkort för att måla ut grafik krävs att spellogik och rendering sker parallellt med varandra för att utnyttja hårdvaran till fullo. Varje tidssteg går genom en pipeline (Žilys 2020). Samtidigt som spellogiken för ett tidssteg beräknas av CPUn renderas bildrutan för föregående tidssteg på GPUn, se figur 4.

När utnyttjningsgraden är hög minskar pipelining den genomsnittliga latensen eftersom datorn hinner beräkna fler tidssteg under samma tidsperiod när uppdatering av spellogik och rendering sker parallellt (Žilys 2020). Utnyttjningsgraden är alltid hög när trippelbuffrad rendering används eller när vertikal synkronisering är avstängd, eftersom datorn då renderar ut så många bildrutor den hinner med utan begränsning.

Däremot när utnyttjningsgraden är låg, som när dubbelbuffrad vertikal synkronisering används i ett mindre krävande spel, kan pipelining öka latensen eftersom mycket av tiden i varje steg i pipelinen används på att vänta på att nästa steg ska färdigställas, se figur 4. Den interna latensen i spelmotorn blir då så hög som antalet pipeline-steg dividerat med bildrutefrekvensen (Žilys 2020). Den blir ännu högre om en renderingskö används för att låta processorn räkna i förväg. Om bildrutefrekvensen är 60 FPS och spelmotorn har två pipeline-steg (uppdatera spellogik, rendera) blir då den interna latensen i spelmotorn minst 33 ms, oavsett hur snabb själva uppdateringen av spellogiken eller renderingen är.

2.3.1 Separat renderingstråd i drivrutin

För att uppnå parallellism mellan CPUn och GPUn genom pipelining krävs en separat renderingstråd som styr GPUn parallellt med att huvudtråden beräknar framtida bildrutor. OpenGL-standarden är designad för att delegera hantering av denna separata renderingstråd till grafikdrivrutinen. APIn är designad så att alla API-anrop kan användas som om de vore synkrona, men så länge detta garanteras är implementationen fri att utföra dem asynkront bakom kulisserna för att möjliggöra parallellism (OpenGL Wiki contributors 2021). Eftersom den övervägande majoriteten av dataflödet är från CPUn till GPUn och inte åt andra hållet fungerar detta oftast bra. Dock är det viktigt att programmeraren är medveten om detta och inte av misstag använder API-anrop som kräver synkronisering. Även i de fall data behöver föras över åt andra hållet finns mekanismer i OpenGL att hämta data asynkront, för att inte begränsa parallellismen (OpenGL Wiki contributors 2019).

Det är värt att understryka att asynkron exekvering av API-anrop inte är något som garanteras av OpenGL-standarden själv. Det är upp till varje implementation att bestämma hur och när API-anropen körs (OpenGL Wiki contributors 2019). Det är även implementationens ansvar att API-anropen körs på ett sätt som tillmötesgår OpenGL-standarden. Moderna drivrutiner brukar dock utföra alla API-anrop asynkront i den mån det går för att uppnå bästa möjliga prestanda.

2.4 Tidigare forskning

2.4.1 Latensens påverkan på prestation i spel

Det finns gott om vetenskapligt stöd för att latens har en betydande påverkan på prestation i spel. Forskare från Nvidia har vid tre tillfällen under de tre senaste åren publicerat journalartiklar där relationen mellan latens och spelarnas prestanda undersöks. I den första av dessa artiklar påvisas ett tydligt samband mellan högre latens och sämre spelarprestanda, och forskarna kommer dessutom fram till att uppdateringsfrekvens har en relativt liten påverkan i de flesta fall när den minskade latensen som en högre uppdateringsfrekvens medför elimineras (Spjut et al. 2019).

I den andra artikeln utforskas hur liknande latenskompensationsteknik som den som används i VR även kan användas i icke-VR spel och öka spelarprestanda. Tekniken som används kallas ”*post-render warp*” och innebär att färdigrenderade bildrutor projiceras om precis innan de visas på skärmen för att ge illusionen av snabbare responstid för rotation. Studien kommer fram till att tekniken förbättrar prestation i system med hög latens, jämfört med att inte använda någon kompensation alls (Kim, Knowles, Spjut, Boudaoud & McGuire 2020).

Båda dessa artiklar är referentgranskade och för den sista har det även utvecklats en interaktiv webbdemo som använder sig av tekniken för att låta användaren utföra ett liknande experiment på egen hand (Boudaoud, Knowles, Kim & Spjut 2021). Den tredje och senaste artikeln stödjer också att latens spelar stor roll och hittar skillnader i hur lång tid det tar att slutföra en uppgift även mellan så lite som 12 ms och 20 ms systemlatens (Spjut, Boudaoud & Kim 2021, preprint). Det är dock värt att notera att denna artikel är publicerad som preprint när detta skrivs [24-05-22] och är därmed inte referentgranskad. Det är även värt att notera att alla dessa artiklar är publicerade av Nvidia som i samband med att de släppt dessa journalartiklar även börjat fokusera mycket på systemlatens i sina produkter och sin marknadsföring. Därmed finns det en inneboende intressekonflikt eftersom det är ekonomiskt fördelaktigt för Nvidia att påvisa att latens påverkar prestation i spel.

Dock är inte forskare från Nvidia de enda som undersökt hur latens påverkar prestation i spel. En studie av Liu, Claypool, Kuwahara, Scovell och Sherman (2021) undersökte hur latens påverkade erfarna Counter Strike: Global Offensive-spelare och kom fram till att både spelarprestation och den upplevda kvaliteten av spelupplevelsen ökade vid lägre latens. I en annan tidigare studie av Claypool, Cockburn och Gutwin (2020) diskuteras, utöver att latens spelar stor roll för spelarprestation, även hur möjligheten att följa olika typer av rörelse med en datormus påverkas av latens.

2.4.2 Nvidia Reflex och minimering av den lokala latensen i dataspel

Nvidia Reflex är en mjukvaruteknologi vars mål är att minska latens genom att dynamiskt fördröja när spellogiken uppdateras så inmatning läses av så nära renderingen som möjligt (Schneider 2020). Reflex inkluderar även andra hårdvaruspecifika justeringar för att optimera latens på andra vis (ibid.). Scheider (2020) förklarar att Reflex är ett resultat av Nvidias forskning inom e-sport. Tyvärr har denna forskning inte publicerats av Nvidia än (Nvidia Corporation u.å.) annat än i form av marknadsföringsmaterial inklusive nyhetsinlägget skrivet av Schneider (2020). Exakta algoritmer eller tekniker som undersöktes för att utveckla teknologin finns därför inte tillgängligt. Utöver Nvidia Reflex finns det en klar avsaknad på forskning inom detta område. Därför fanns det inte mycket vetenskaplig information att utgå från i detta arbete.

3 Problemformulering

Dubbelbuffrad vertikal synkronisering har många önskvärda egenskaper, inklusive jämn rörelse, ingen tearing och god energieffektivitet. Dess höga latens gör det dock opraktiskt att använda i spel där responstid är viktigt, exempelvis e-sportspel och musikspel. För spel inom dessa kategorier som inte är särskilt prestandakrävande innebär det att beräkningarna i varje pipeline-steg hinner bli klara i god tid och mycket av tiden spenderas på att vänta in nästa steg i pipelinen när dubbelbuffrad vertikal synkronisering används.

En lösning till detta är att fördröja när spellogiken uppdateras på ett liknande sätt som Nvidia Reflex gör. Då sker väntetiden till innan beräkningarna snarare än efter dem. Det gör att inmatning kan hämtas senare och därmed minskar spelmotorns interna latens. Utifrån egen testning i Fortnite (Epic Games 2017) verkar dock inte Nvidia Reflex fungera tillsammans med vertikal synkronisering. Detta bekräftas av Battle(non)sense (2020) som kommer fram till samma resultat och förklarar att Nvidia Reflex enbart hjälper när grafikkortet är flaskhalsen, vilket det inte är när vertikal synkronisering används.

Att fördröja när spellogiken uppdateras för att minska latens är en teknik som inte verkar ha något etablerat universellt namn än. I denna rapport föreslås och används termen ”JIT-baserad latensreduktion”, eller ”just-in-time-baserad latensreduktion” för att beskriva tekniken. Namnet är inspirerat av JIT inom tillverkning, vilket är en strategi för att minska produktionstiden och inventering (Banton 2021). Det finns många paralleller mellan JIT och tekniken som utforskas i detta arbete, däribland krav på tillförlitliga prognoser på framtida efterfrågan och större risk för förseningar (ibid.). Även Schneider (2020) använder termen ”just-in-time” för att beskriva hur Nvidia Reflex fungerar.

Deadlinen som den JIT-baserade latensreduktionen i Nvidia Reflex förhåller sig till är att ett tidssteg ska vara färdigberäknat på CPUn precis när GPUn är redo att rendera det. Deadlinen som JIT-baserad latensreduktion för dubbelbuffrad vertikal synkronisering behöver förhålla sig till är att ett tidssteg ska vara både färdigberäknat på CPUn och färdigrenderat på GPUn precis innan det vertikala blankningsintervallet. Den största skillnaden mellan dessa deadlines är att den sistnämnda vanligtvis inte är flexibel. Missas det vertikala blankningsintervallet behöver nästa vertikala blankningsintervall inväntas, vilket ställer högre krav på att deadlinen missas så sällan som möjligt för att undvika lagg.

Målet med detta arbete var att implementera JIT-baserad latensreduktion för dubbelbuffrad vertikal synkronisering och besvara följande två frågor genom experiment: Är det praktiskt och gynnsamt att använda JIT-baserad latensreduktion i spel med dubbelbuffrad vertikal synkronisering? Hur ska prognosen för framtida beräkningstid tas fram för att ge en bra avvägning mellan latens och stabilitet vid olika belastningar?

3.1 Metodbeskrivning

3.1.1 Prognos av beräkningstid

Det finns ett viktigt problem som behövde lösas för att JIT-baserad latensreduktion skulle kunna implementeras: Hur vet spelmotorn hur länge den kan vänta och fortfarande hinna klart i tid? Egentligen behöver spelmotorn veta hur lång tid det kommer ta att beräkna spellogiken och utföra rendering för nästa bildruta innan det har skett, vilket är omöjligt. Det krävs därmed någon form av prognosberäkning för att åstadkomma detta.

Om den verkliga beräkningstiden är högre än den förutspådda, exempelvis om prestandakraven plötsligt ökar mer än väntat, kommer beräkningarna fördröjas för länge och det vertikala synkroniseringsintervallet missas. Det innebär att nästa vertikala synkroniseringsintervall behöver inväntas innan buffertbytet sker, vilket kommer upplevas som lagg. Det är därför kritiskt att prognosberäkningen sällan underskattar tiden, vilket ställer höga krav på prognosberäkningens förmåga att anpassa sig efter variationer i prestandakrav.

I denna implementation av JIT-baserad latensreduktion tas prognosen fram genom att ta beräkningstiden från ett antal tidigare bildrutor, ta medelvärdet eller medianen av dessa och slutligen lägga på en säkerhetsbuffert. Parametervärdena som används i denna beräkning är kritiska för att prognoserna ska vara tillräckligt låga för att skapa en genomsnittlig minskning av latensen, utan att vara så låga att det ofta sker underskattningar av beräkningstiden.

3.1.2 Experimentet

För experimentet användes en prestandatestapplikation som simulerar ett partikelsystem. Varje partikel består av testmodellen Stanford Bunny. Versionen som används är från Computer Graphics Archive och har 144 046 trianglar (McGuire 2017). Dess höga triangelantal gjorde den passande för att belasta grafikprocessorn.

Shaderprogrammet som renderar varje partikel förflyttar modellen med transformations-vy-perspektiv-matrisen och färglägger modellen med hörn-normal-vektorn. Partiklarna roteras och förflyttas med en gravitationskraft. Partiklarna skapas vid fasta intervall över tid eller med slumpade intervaller och har en livstid på 2 sekunder. I testapplikationen går det att specificera det genomsnittliga antalet partiklar, huruvida de skapas med jämt mellanrum eller vid slumpade intervaller samt parametervärdena som används för prognos av prestanda.

Experimentet hade två syften. Det första var att undersöka hur olika belastningar påverkar prestandan och latensen när applikationen använder sig av olika parametervärden för latensreduktionens prognos av beräkningstid. Poängen med detta var att förstå vilka parametervärden som ger en god avvägning mellan prestanda och latens vid de olika belastningarna. Det andra var att undersöka hur prestanda och latens jämför sig mellan när latensreduktion med passande parametervärden används och när det inte används.

Parametervärdena för prognosen som testades var alla kombinationer av följande:

- Antal tidigare tidssteg: 1, 2, 3, 4, 5
- Hur grundvärdet extraheras från tidigare tidssteg: medelvärde, median
- Hur stor säkerhetsmarginalen är: 2 ms, 3 ms, 4 ms, 5 ms, 6 ms, 7 ms

Parametervärdena för antal tidigare tidssteg valdes för att värdet behöver läsas ut från minst 1 bildruta men mer än 5 bildrutor hade i teorin gett dålig responstid på latensreduceringens förmåga att anpassa sig till ändrade prestandakrav. Parametervärdena för säkerhetsmarginalen valdes eftersom en lägre säkerhetsmarginal än 2 ms hade garanterat gett prestandaproblem, eftersom schemaläggarens högsta precision är 2 ms, och en högre säkerhetsmarginal än 7 ms gav minskande avkastning när preliminär testning av tekniken utfördes.

Parametervärdena för prognosen testades under de belastningar som beskrivs av alla kombinationer av följande:

- Det genomsnittliga antalet partiklar: 250, 750
- Hur partiklarna skapas: Med jämna mellanrum, vid slumpade intervaller

Det genomsnittliga antalet partiklar valdes eftersom 750 partiklar är nära den övre gränsen av vad datorn som experimentet kör på klarar av utan lagg, och 250 partiklar är långt under denna gräns. När partiklarna skapas med jämna mellanrum kollas vid varje tidssteg hur många partiklar som borde skapats mellan förra tidssteget och detta, sedan skapas så många partiklar. När partiklarna skapas vid slumpade intervaller adderas det antalet partiklar till i en variabel, i stället för att skapas direkt. Varje tidssteg är det ¼ chans att det skapas lika många partiklar som värdet i denna variabel samt att variabeln återställs till 0. Detta är för att efterlikna prestandavariationerna som kan uppstå i spel när objekt regelbundet skapas eller förstörs.

Metoden för detta experiment behövde vara mer åt det utforskande hållet eftersom det inte fanns någon tidigare data att utgå från. Valet gjordes därför att samla ihop data för alla kombinationer av parametervärden, snarare än att bara justera ett parametervärde åt gången, vilket gjorde det möjligt att både observera hur flera faktorer samspelar med varandra samtidigt som enskilda faktorer kunde isoleras. Denna multidimensionella datainsamling valdes för att ge en bättre helhetsbild av hur tekniken presterar. Dock gjorde den insamling, analys, visualisering och presentation av data mer otymplig. Därför var det viktigt att noggrant välja ut vilka parametervärdena som testades för att ge så värdefulla data som möjligt. Sammanlagt utgör detta experiment 240 olika körningar av testapplikationen. Det hade varit önskvärt att testa fler variationer av antal partiklar och typer av slumpgenerering, men detta hade flerdubblat antalet körningar av testapplikationen.

Parametervärden för latensreduceringens prognos och belastningen går att ställa in genom att köra programmet med argument i kommandotolken. Det möjliggjorde automatisering av testkörningen via ett batch-skript. Skriptet startar testet, väntar på att det avslutar och kör därefter nästa test, till dess att alla tester är körda.

3.1.3 Validitet

Ett inneboende problem av att använda en artificiell testbelastning för detta experiment är att resultatet inte nödvändigtvis blir representativt av ett faktiskt spel. Testbelastningen som används i detta arbete belastar dessutom huvudsakligen grafikkortet, vilket är fallet i många spel men långt ifrån alla. Hade däremot en testbelastning som huvudsakligen belastar processorn också implementerats hade det fyrdubblat mängden testresultat att hantera, så det lämnas åt framtida forskning.

Induktion, alltså att göra generaliseringar baserat på insamlade data, är ett enligt Robert Nola och Howard Sankey (2007/2014) ett koncept inom forskning som är både viktigt och kontroversiellt. Det är viktigt för att kunna applicera kunskap baserad på empiriska data i nya situationer (ibid., ss. 106–107) men det är kontroversiellt eftersom det förutsätter att vi lever i en värld där induktiva regler är pålitliga, vilket är omöjligt att garantera (ibid., ss. 63–64). Datavetenskap skiljer sig här från andra typer av vetenskap eftersom datorer är avsiktligt deterministiska, och det går därför i teorin att förstå allt om hur ett datasystem fungerar. Därmed går det att göra induktiva ansatser som är 100% pålitliga, så länge alla förutsättningar som kan skilja sig i andra typer av datasystem, inklusive framtida datasystem, specificeras. I praktiken är dock dagens datasystem för komplexa för att enskilda personer ska kunna bilda

en komplett förståelse av dem, men det går i alla fall att förstå alla komponenter som är relevanta och därmed uppnå liknande nivåer av pålitlighet.

Syftet med detta experiment är dock inte att komma fram till en generaliserbar teori, utan att göra en första utvärdering av huruvida tekniken har potential att göra nytta i spel och samla in intressant data som kan agera som utgångspunkt för framtida forskning. Det är givetvis viktigt att vara medveten om de mönster som finns i faktiska spel för att de slutsatser som dras ska kunna relateras till spel, exempelvis att beräkningstid varierar något mellan varje tidssteg, men i detta fall är det inte kritiskt att modellen för variation av beräkningstid stämmer exakt överens med riktiga spel. En induktiv ansats utifrån detta arbete hade därför hållit sig på den försiktiga sidan: Om tekniken kan minska latens och bibehålla god prestanda i en spelliknande applikation, under förutsättning att de koncept som beskrivs i del 2 gäller och de begränsningar som beskrivs i del 4.4 tas i åtanke, finns det möjlighet att andra spel eller spelliknande applikationer hade kunnat dra samma nytta av tekniken.

3.1.4 Hårdvara och mjukvara

Experimentet kommer utföras på en stationär dator med processorn Intel Core i5-4690k @ 3.5 GHz och grafikprocessorn Nvidia RTX 2060 av modellen ASUS DUAL-RTX2060-O6G-EVO med 6GB VRAM. Processorn är inte överklockad och grafikkortet är inte överklockad ytterligare utöver fabriksöverklockningen. RAM-minnet i datorn består av 16 GB DDR3 @ 1600 MHz varav två moduler är 4 GB och kör i dual-channel och en modul är 8 GB. Datorn använder operativsystemet Windows 10 Home (Microsoft 2015) version 21H2 och använder energischemat ”Hög prestanda” (se del 4.4). Grafikdrivrutinen är Nvidia Game Ready Driver (Nvidia Corporation 2022) version 512.77 med alla inställningar ställda till sina standardvärden. För att minska risken för att bakgrundsprocesser påverkar resultatet kommer datorn startas om och därefter alla bakgrundsprocesser som inte är inbyggda i Windows att avslutas via aktivitetshanteraren innan experimentet börjar. Dessutom kommer nätverk, Bluetooth och andra enheter vara bortkopplade. Det enda som är kopplat till datorn under mätning är ett trådanslutet tangentbord, en trådansluten mus och en dataskärm med upplösningen 1080p och bildfrekvensen 60 Hz kopplad till en av grafikkortets HDMI-utgångar.

Valet att avsluta bakgrundsprocesser och ha så få enheter som möjligt kopplade till datorn under experimentet är en avvägning mellan validitet och reproducerbarhet. En vanlig användare brukar inte koppla ur sitt internet och avsluta alla andra dataprogram och deras bakgrundsprocesser innan de spelar ett spel, men i detta fall är det nödvändigt för att minska risken för extern påverkan av resultatet. Därför är det värt att ha i åtanke att den påverkan som dessa bakgrundsprocesser hade haft på ett verkligt användningsscenario inte reflekteras i resultatet.

3.1.5 Mätning

Mätning av latens och prestanda sker internt i spelmotorn. Prestandatestapplikationen samlar in mätningarna varje tidssteg och sparar dem till en fil när testet är över. Latens mäts som tiden mellan att inmatning läses av på huvudtråden till det att buffertbytet skett på renderingstråden. Prestanda kommer mätas i form av tiden mellan varje bildrutas buffertbyte på renderingstråden och presenteras både form av genomsnitt och i form av värdenas distribution genom att även presentera genomsnittet av de 10% högsta och 1% högsta värdena. Distributionen avslöjar nämligen hur ofta det uppstår laggspikar, alltså när enstaka bildrutor

tar längre tid än genomsnittet, vilket enligt Scott Wasson (2011) ger en mer representativ bild av den resulterande spelupplevelsen.

Vid varje test samlas data från 500 bildrutor in. Innan datainsamling påbörjar körs de första 120 bildrutorna utan att samla in data. Syftet med detta är att undvika att ”uppvärmningslagg”, alltså det lagg som vanligtvis uppstår precis vid applikationsstart, påverkar resultatet eftersom det inte är representativt för applikationens prestanda i längden. Det gör också att när data börjar samlas kommer det ha skapats så många partiklar som specificerades för testscenariot, eftersom partiklarna har en livstid på 2 sekunder och skärmen som testet utfördes på har bildfrekvensen 60 Hz.

Anledningen att ingen extern mätning av latens baserad på musinmatning görs är för att det kräver verktyg som är dyra eller svåra att få tillgång till, exempelvis ett dedikerat verktyg som mäter latensen eller en kamera med mycket hög bildrutefrekvens. Den kamera med högst bildrutefrekvens som finns tillgänglig för detta arbete kan spela in i upp till 120 bilder per sekund, vilket ger en felmarginal på ±8.3 ms. Mätvärdena väntas vara inom tiotal millisekunder från varandra, i vissa testfall betydligt lägre, så felmarginalen är för hög för att kunna ge ett pålitligt resultat.

Eftersom mätning av latens sker internt i spelmotorn tas inte andra faktorer som påverkar den totala latensen för systemet i åtanke, inklusive uppdateringsfrekvensen för inmatningsenheter, skärmens latens, latens i operativsystemet och liknande. Fördelen är att mätningen då isoleras enbart till komponenten i fråga. Nackdelen är att interna mätverktyg i programmet som mäts medför en större opålitlighet än vad externa mätverktyg gör, eftersom det förlitar sig på att mätverktygen konstrueras korrekt. Som Nola och Sankey (2007/2014, s. 13) beskriver är observation en medveten aktivitet som riskerar att påverka resultatet, och eftersom att utföra en intern prestandamätning har en viss prestandakostnad i sig kan interna prestandamätningar artificiellt öka mätvärdena. I moderna datorer är dock prestandakostnaden av att utföra dessa mätningar så låg att den i de flesta fall är försumbar. Den interna mätningen bör vara tillräckligt tillförlitlig för att kunna användas för att dra rimliga slutsatser om hur den genomsnittliga latensen för hela systemet påverkas. Dock hade en extern mätning utöver de interna varit önskvärd.

3.1.6 Hypotes

Utifrån problemformuleringen togs en hypotes fram som utgångspunkt för experimentet. Hypotesen bestod av följande förväntningar:

1. Den genomsnittliga latensen väntas minska när JIT-baserad latensreduktion används jämfört med när det inte används. Dock väntas risken för lagg öka när JIT-baserad latensreduktion används.
2. Att öka prognosens säkerhetsmarginal väntas minska risken för lagg, men väntas öka den genomsnittliga latensen.
3. JIT-baserad latensreduktion väntas minska latens mer vid lätta belastningar än vid tyngre belastningar.
4. Instabila belastningar väntas kräva högre säkerhetsmarginal för att undvika lagg.
5. Att använda medianen av beräkningstiden för ett antal tidigare bildrutor i prognosen väntas ge mindre lagg än att använda medelvärdet av dessa.

6. Att använda mätvärden från ett för lågt antal bildrutor i prognosen väntas öka risken för lagg generellt, men att använda mätvärden från ett för högt antal bildrutor väntas öka risken för lagg när beräkningstiden varierar mycket över tid.

Punkt 1–4 är grundade i det som diskuterats tidigare i texten. Punkt 5–6 behandlar prognosens anpassningsförmåga i föränderliga omständigheter, vilket väntas vara en viktig egenskap för att hantera instabila belastningar på ett passande sätt.

4 Implementation

4.1 Spelmotorn

JIT-baserad latensreduktion implementerades i en spelmotor vid namn Kamera Engine. Det är en egenutvecklad spelmotor som är skriven i C++ 14 (ISO/IEC 14882:2014) och stödjer 64-bitars versionen av Windows 10 (Microsoft 2015). Den är i ett tidigt stadie men har sedan tidigare stöd för att skapa ett fönster och läsa av inmatningshändelser från operativsystemet via biblioteket GLFW (Geelnard & Löwy 2021) samt rita ut 3D-modeller på skärmen med OpenGL 4.3 (Segal & Akeley 2013) genom en *loader* från GLAD (Herberth 2021). Spelmotorn läser in 3D-modeller med hjälp av biblioteket tinyobjloader (Fujita 2019) och texturer med hjälp av biblioteket stb_image (Barrett 2017). Den har även begränsat stöd för serialisering med hjälp av biblioteket yaml-cpp (Beder 2015), men detta används inte i arbetet. Mattebiblioteket som används heter KaMath och är egenskrivet specifikt för spelmotorn.

Kamera Engine har ännu inte någon färdig ”editor”-applikation för att skapa och bygga spelOprojekt i. För att använda spelmotorn behöver klientapplikationen länka statiskt mot Kamera Engine, KaMath och övriga bibliotek som Kamera Engine är beroende av. Det finns en testapplikation som utnyttjar spelmotorns funktioner och demonstrerar att de fungerar. Denna testapplikation använder även GUI-biblioteket Dear ImGui (Cornut 2022) för att visa statistik och exponera spelmotorinställningar. Testapplikationen kunde utnyttjas för att snabbt testa implementationen av latensreduceringen.

Kamera Engine valdes eftersom den utgjorde en god grund att utföra arbetet utifrån. Det är en egenutvecklad spelmotor som är i ett tidigt stadie, så ingen tid behövde läggas på att förstå hur en annan mer komplex spelmotor fungerar. Det hade annars kunnat bli en risk eller flaskhals för arbetet. Nackdelen med att använda Kamera Engine var att vissa saker saknades och behövde implementeras för att arbetet skulle gå att slutföra.

4.1.1 Förarbete

För att arbetet skulle gå att slutföra behövde vissa latens- och prestandamätningsverktyg implementeras i Kamera Engine. Det var viktigt att implementera mätning av spelmotorns totala latens så tidigt som möjligt för att snabbt kunna avgöra att latensreduktion fungerade korrekt. Dessutom behövde det implementeras ett sätt att utföra tidmätningar på grafikdrivrutinens renderingstråd för att kunna mäta GPU-prestanda och spelmotorns totala latens.

Tidmätningar på GPUn använder sig av OpenGL queries, som är ett sätt att ställa en ”fråga” till grafikdrivrutinen och hämta svaren först när de finns tillgängliga (OpenGL Wiki Contributors 2019). Runt denna funktionalitet implementerades två klasser, ”RenderQueryTimePoint” för att mäta tidpunkten precis efter ett renderingskommando färdigställts som en tidpunkt hos systemklockan och ”RenderQueryTimer” för att mäta tidsspannet mellan när två renderingskommandon färdigställs.

Latensmätningen i spelmotorn implementerades för enkelhetens skull direkt i Applikations-klassen. De fungerar genom att mäta tidpunkten när en bildruta startar på huvudtråden och tidpunkten när en bildruta färdigställts på renderingstråden och sedan ta differensen mellan dessa. Eftersom tidmätningen på renderingstråden inte är tillgänglig omedelbart när frågan

skickas färdigställs resultaten allt eftersom de blir tillgängliga. Latensmätningarna använder en tidsstegsräknare för att hålla reda på vilken mätning som är vilken.

För att undvika att själva latensmätningen påverkas av de minnesallokeringar och andra beräkningar som utförs av mätverktygen själva är det viktigt att själva tidmätningen sker vid rätt tillfälle. I början av en mätning bör mätningen ske så sent som möjligt och i slutet av en mätning bör mätningen ske så tidigt som möjligt. På så sätt hamnar verktygets egna beräkningar utanför mätningen. Alla tidmätningsverktyg i detta arbete konstruerades med detta i åtanke.

4.2 Latensreduceringsbiblioteket KaJit

För att göra koden för JIT-baserad latensreducering återanvändbar, så den går att implementera i andra program, samlades denna kod i ett eget C++-bibliotek vid namn KaJit. Detta bibliotek implementerades i sin tur i Kamera Engine. I denna del beskrivs hur KaJit fungerar och implementeras.

4.2.1 Prestandaövervakaren

KaJit innehåller ett prestandaövervakningsverktyg som bygger på samma princip som latensmätningsverktyget i Kamera Engine. Den största skillnaden är att prestandaövervakningsverktyg samlar alla mätningar i de tidssteg som de tillhör och sparar gamla mätningar en stund för att möjliggöra att använda data från flera tidssteg.

Det finns två sätt att utföra mätningar med prestandaövervakningsverktyget i KaJit. Det första är att använda prestandaövervakarens egna funktioner ”StartMeasurement” och ”StopMeasurement” för att starta och stoppa prestandamätningar på huvudtråden. Det andra är att lägga till mätvärden manuellt i prestandaövervakaren vilket gör att hur mätningen sker kan definieras hos klientapplikationen. Asynkrona mätningar kan läggas till i prestandamätningsverktyget genom att skapa en klass baserat på den abstrakta basklassen ”AsynchronousTimer” som deklarerar funktionen ”TryGetResult”. Denna funktion ska returnera sant och sätta resultatet i den första parametern om det finns tillgängligt och annars returnera falskt. Asynkrona mätningar läggs till genom att kalla funktionen ”AddAsynchronousTimer” med en pekare till en klass som ärver av ”AsynchronousTimer”, som definieras av klientapplikationen. Genom denna mekanism kan prestandaövervakaren läsa av mätvärdet av asynkrona mätningar först när de blir tillgängliga på ett sätt som är oberoende av klientapplikationens implementation av asynkrona prestandamätningar. Implementationen i Kamera Engine är en wrapper-klass kring ”RenderQueryTimer”-klassen.

4.2.2 Renderingsövervakaren

Tyvärr finns det inget sätt att ta reda på när det vertikala blankningsintervallet kommer ske nästa gång. Det finns inte heller något inbyggt sätt att ta reda på när det senaste vertikala blankningsintervallet skedde. Det går dock att arbeta runt genom att göra en tidmätning direkt efter att skärbuffertarna bytt plats, vilket ger en god estimering av när det vertikala blankningsintervallet skedde senast. Utifrån denna tidmätning och skärmens uppdateringsfrekvens går det i sin tur att extrapolera när nästa vertikala blankningsintervall väntas ske.

Renderingsövervakaren har hand om att spara mätningar av när varje vertikalt blankningsintervall sker. Precis som prestandaövervakaren tar funktionen

SetAsynchronousReferenceVBI” en pekare till en abstrakt basklass vid namn ”AsynchronousTimePoint” som parameter, för att möjliggöra implementationsoberoende användning av asynkrona tidmätningar. Implementationen av ”AsynchronousTimePoint” i Kamera Engine är en wrapper-klass kring ”RenderQueryTimePoint”-klassen.

4.2.3 Tajming med hög precision i ett icke-realtidsoperativsystem

För att fördröja när inmatning hämtas behöver huvudtråden blockeras under en specifik tidsperiod med relativt hög precision, helst inom några millisekunder. Det enklaste, mest interkompatibla och mest precisa sättet att lösa detta på är genom en så kallad ”busy wait” där tråden går in i en loop som gör ingenting tills den specificerade tidsperioden passerat. Nackdelen är att detta är väldigt ineffektivt eftersom det slösar processortid som operativsystemet hade kunnat använda till något annat (Silberschatz, Galvin & Gagne 2008, ss. 236). Framför allt är det opassande för batteridrivna enheter vars batteritid hade påverkats negativt av en sådan slösaktig lösning.

Ett mer energieffektivt sätt att lösa detta på är genom schemaläggningsbaserad blockering. Då bestämmer operativsystemets schemaläggare hur väntetiden ska användas. Dock kommer det med sina egna problem. Som standard är Windows schemaläggare väldigt oprecis eftersom den endast schemalägger med en frekvens på 64 hz vilket ger en precision på 15.6 ms (Dawson 2020), vilket är alldeles för oprecist för att vara användbart för detta syfte. Dock går detta att kringgå genom att använda så kallade ”waitable timers” och temporärt höja schemaläggarens precision. Genom denna metod går det att blockera tråden med en precision på ner till 2 ms (ibid.).

Det är dock viktigt att notera att Windows inte är ett realtidsoperativsystem. Därför finns inga garantier för när operativsystemet återupptar exekvering av schemalagda processer, annat än efter den tidsperiod som specificeras (Silberschatz, Galvin & Gagne 2008, ss. 764–765). Om andra processer hamnar före i kön, exempelvis processer med högre prioritet, kommer schemaläggaren låta dem exekvera först. Därmed kan det hända att exekvering inte återupptas förrän långt efter tiden som specificerats. Tiden som specificeras vid schemaläggningsbaserad blockering bör alltså enbart ses som ett önskemål om när processen ska schemaläggas nästa gång, inte en deadline.

Som tur är prioriterar Windows trådar tillhörande förgrundsprocessen som avslutar en ”wait”-operation högt (Russinovich & Solomon 2009, s. 419). Dock ska det tilläggas att schemaläggningsproblem inte är exklusivt till processer som frivilligt lägger sig i sömnläge. Windows schemaläggare är ”preemptive” (ibid., s. 391), vilket innebär att den när som helst kan avbryta en process som tar upp mycket processortid för att låta andra processer utföra arbete samtidigt[5] (Silberschatz, Galvin & Gagne 2008, s. 765). Allt som allt utgjorde detta inte ett hinder för arbetet, men det påverkar hur höga säkerhetsmarginaler som krävs för att uppnå ett stabilt resultat. KaJit implementerar schemaläggningsbaserad blockering med en precision på 2 ms i funktionen ”HighPrecisionWait”, som i sin tur används vid latensreducering.

4.2.4 Latensreducering

KaJit har en klass ”LatencyReducer” som har en prestandaövervakare och en renderingsövervakare. När dessa förses med relevanta prestandamätningar kan klassens

[5] Notera skillnaden mellan samtidighet och parallellism.

funktion ”ReduceLatency” anropas precis före inmatning hämtas för att förskjuta när inmatningen sker. Funktionen tar emot följande parametervärden:

- Latensreduceringsinställningar: Parametervärden för hur prognosen ska beräknas.
- Tid mellan VBI: Hur lång tid det är mellan varje vertikalt blankningsintervall, alltså $\frac{1}{n}$ där n är skärmens uppdateringsfrekvens.
- Renderingskös storlek: Hur många bildrutor som är i renderingskön när funktionen anropas (inklusive eventuell bildruta som renderas vid detta tillfälle).

Antalet bildrutor i renderingskön beräknas i Kamera Engine genom att använda asynkrona tidmätningar. Precis innan huvudtråden börjar skicka renderingskommandon ökas värdet på en räknarvariabel med ett. En asynkron tidmätning sker precis efter buffertbytet skett. När resultatet av denna tidmätning är tillgänglig minskas värdet på räknarvariabeln med ett.

Funktionen ”ReduceLatency” anropar i sin tur en privat funktion ”CalculateLatencyReductionDelay" med samma parametervärden och använder returvärdet från denna funktion som parameter när den anropar ”HighPrecisionWait”. Beräkningen som sker i ”CalculateLatencyReductionDelay” beskrivs i del 4.3.

4.2.5 Implementation i andra applikationer

Nedan är en lista för att sammanställa vad som behöver göras för att implementera KaJit i en applikation (förutom länkning). KaJit har inga beroenden utöver de Windows-bibliotek som finns tillgängliga som standard i Visual Studio 2019. Alla klasser och funktioner är i namnrymden ”KaJit”.

- ”LatencyReducer::Initialize” och ”LatencyReducer::Terminate” behöver anropas vid applikationsstart respektive applikationsavslut.
- Mätning av CPU-tid och GPU-tid behöver förses till prestandaövervakaren varje tidssteg. För asynkrona mätningar behöver en klass som ärver av ”AsynchronousTimer” implementeras.
- Mätning av VBI behöver förses till renderingsövervakaren varje tidssteg. För asynkrona mätningar behöver en klass som ärver av ”AsynchronousTimePoint” implementeras.
- Storleken på bildrutekön behöver beräknas och hållas uppdaterad.
- Funktionen ”ReduceLatency” behöver anropas varje tidssteg precis innan inmatning hämtas.
- Applikationen behöver antagligen vara i exklusivt helskärmsläge och använda ett grafikkort som är kopplat direkt till skärmen. Mer information om detta finns i del 4.4.

4.3 Prognos av beräkningstid

För att kunna fördröja när inmatning hanteras är det kritiskt att veta hur mycket tid som går att fördröja med och fortfarande hinna med alla beräkningar innan det vertikala blankningsintervallet. Att beräkna denna fördröjning består av tre delar. Först behöver det skapas en prognos av den framtida beräkningstiden för detta tidssteg, baserat på prestandamätningar från tidigare tidssteg. Därefter används denna prognos för att ta fram en deadline när tidssteget behöver vara färdigberäknat för att hinna i tid till nästa bästa vertikala blankningsintervall. Slutligen tas en fördröjningstid fram baserat på deadlinen och den uppskattade beräkningstiden.

4.3.1 Estimering av beräkningstid

Beroende på ett parametervärde hämtas beräkningstider från ett visst antal tidigare färdigställda tidssteg. CPU- och GPU-tiderna för dessa tidssteg summeras ihop till respektive tidsstegs totala beräkningstid. Genomsnittet, antingen i form av medelvärde eller median beroende på ytterligare ett parametervärde, tas av både varje tidsstegs totala beräkningstid respektive varje tidsstegs GPU-tid. Dessa genomsnitt blir den estimerade totala beräkningstiden respektive den estimerade GPU-beräkningstiden. Den estimerade CPU-beräkningstiden fås av den estimerade totala beräkningstiden minus den estimerade GPU-beräkningstiden. Anledningen till detta är att korrektheten av den estimerade GPU-beräkningstiden är viktigare än korrektheten av den estimerade CPU-beräkningstiden ifall medianen används. Om medelvärdet används spelar det ingen roll för då blir resultatet det samma oavsett[6].

4.3.2 Beräkning av deadline

För att räkna ut deadlinen används en funktion som heter ”NextVBIAfter”. Den tar in ett parametervärde som är en tid från och med att funktionen anropas och funktionen returnerar när nästa vertikala blankningsintervall efter denna tid väntas ske. Detta estimeras utifrån bildskärmens uppdateringsfrekvens och en tidmätning som sker direkt efter varje vertikalt blankningsintervall.

Om renderingskön är tom beräknas deadlinen genom att anropa NextVBIAfter med den estimerade totala beräkningstiden som parametervärde. Om renderingskön inte är tom beräknas deadlinen i två steg. Först beräknas om hur lång tid renderingstråden väntas vara klar med tidigare bildrutor, r, genom att anropa NextVBIAfter med parametervärdet $\frac{n}{F}$, där n är antalet bildrutor som är kvar och F är bildskärmens uppdateringsfrekvens. Därefter räknas deadlinen ut genom att anropa NextVBIAfter med parametervärdet $\max(r, t_c) + t_g$ där t_c är den estimerade CPU-beräkningstiden och t_g är den estimerade GPU-beräkningstiden. Max-funktionen av r och t_c är där eftersom tidigare bildrutor kan renderas på GPUn parallellt med att ett nytt tidssteg beräknas på CPUn. Deadlinen kommer därför sättas efter det att båda dem väntas vara klara, plus efter att bildrutan renderats av GPUn. Det är här som korrektheten av GPU-beräkningstiden gör större nytta än korrektheten av CPU-beräkningstiden.

4.3.3 Beräkning av väntetid

Beräkning av väntetiden är enkel och görs genom att ta $d - (E + m)$ där d är deadlinen, E är den estimerade totala beräkningstiden och m är ett parametervärde som anger säkerhetsmarginalen. Säkerhetsmarginalens syfte är att kompensera för variationer i den schemaläggningsbaserade blockeringen, oväntade ändringar i prestandakrav och andra små variationer i de tidmätningar som utförs. Anledningen att säkerhetsmarginalen inte används för att beräkna deadlinen är för att den då riskerar att sätta deadlinen för långt fram och skapa lagg i onödan. Det behövs dessutom inte, eftersom renderingsköns storlek redan används för

[6] Om T_c och T_g är den totala beräkningstiden för CPU respektive GPU över ett antal tidssteg n är den genomsnittliga beräkningstiden $\frac{T_c}{n}$ för CPU, $\frac{T_g}{n}$ för GPU och $\frac{T_c+T_g}{n} = \frac{T_c}{n} + \frac{T_g}{n}$ för den totala beräkningstiden. Således går det att beräkna en av dessa genomsnitt utifrån de två andra. Om medianen används finns det risk att medianen för CPU, GPU och totalen är från olika tidssteg, vilket innebär att summan av medianerna för beräkningstiderna för CPU och GPU inte nödvändigtvis är lika med medianen av den totala beräkningstiden.

att avgöra när nästa deadline ska sättas. Det räcker alltså att använda marginalen i slutberäkningen av väntetiden för att åstadkomma önskat resultat.

4.4 Implementationens begränsningar

Eftersom olika operativsystem använder olika schemaläggare krävs plattformsspecifika implementationer för att implementera schemaläggningsbaserad blockering med precision som är hög nog. Som tidigare nämnt är det möjligt att implementera blockering i form av en ”busy wait” på vilken plattform som helst, men för operativsystem som körs på bärbara enheter är detta som tidigare diskuterat inte ett passande alternativ. Finns där inte ett bra sätt att schemalägga med hög precision går det kanske inte att implementera JIT-baserad latensreduktion på ett passande sätt.

En annan begränsning på just Windows är att JIT-baserad latensreduktion enbart fungerar som det ska i exklusivt helskärmsläge, dels för att det inte går att hämta bildskärmens uppdateringsfrekvens från GLFW i fönsterläge, dels för att Windows fönsterhanterare verkar orsaka prestandaproblem som kräver mycket högre säkerhetsmarginal för att uppnå en stabil bildfrekvens med JIT-baserad latensreduktion. Liknande prestandaproblem uppstår även när tekniken testades på en laptop med ett dedikerat grafikkort som använder Nvidia Optimus och saknar en multiplexer, alltså där det dedikerade grafikkortet inte har direkt åtkomst till skärmen utan i stället måste överföra sina bildrutor till den integrerade grafikkretsen som i sin tur skannar ut dem på skärmen (Farncomb 2021). Den gemensamma nämnaren mellan dessa två fall är att den färdigställda bildrutan måste överföras till en annan mjukvaru- eller hårdvarukomponent innan den skannas ut. Detta ökar antagligen osäkerheten på när buffertbytet sker eftersom det då inte längre är direkt kopplat till skärmens vertikala blankningsintervall.

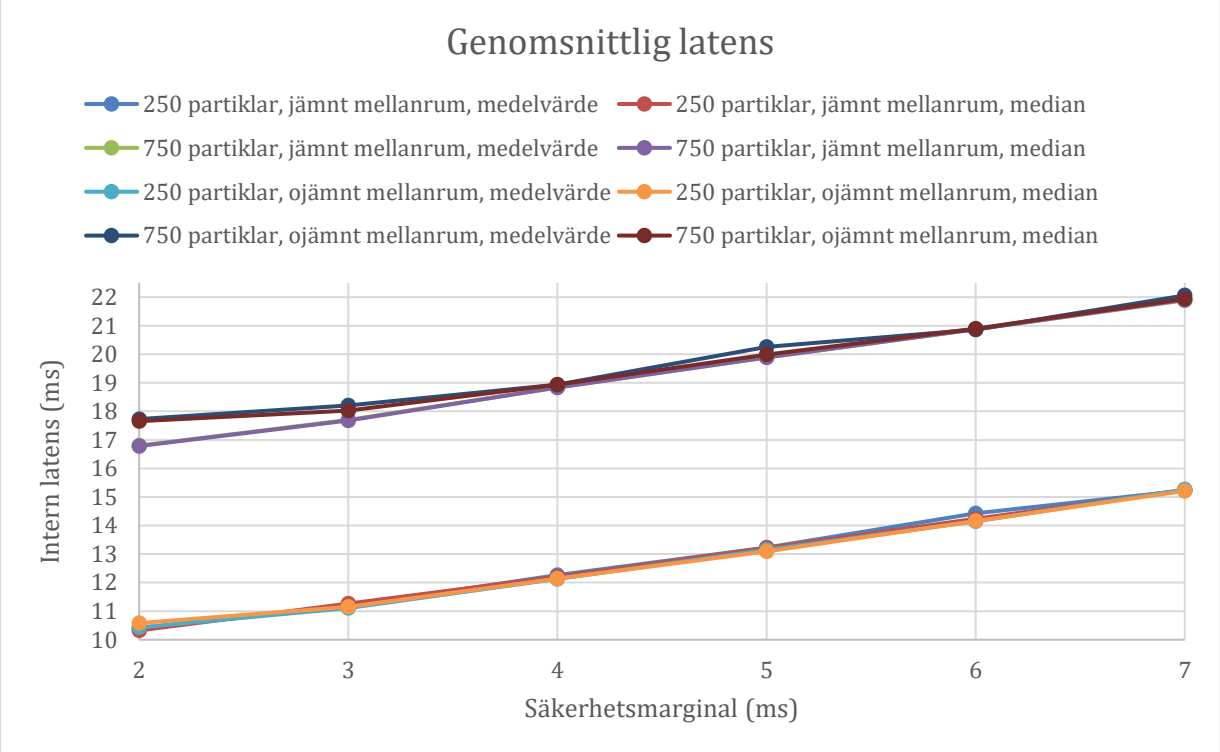
Dessutom gick det inte att implementera så grafikkortets klockhastighet kunde styras av i implementationen, vilket gör att renderingstiden blir beroende av hur grafikdrivrutinen väljer att justera grafikkortets klockhastighet. När JIT-baserad latensreduktion används blir grafikkortets användningsmönster olikt vad som vanligtvis uppstår i spel. Vanligtvis går det att anta att utnyttjningsgraden av grafikkortet har en direkt korrelation med hur mycket tid grafikkortet har på sig att rendera en bildruta. Vid användning av JIT-baserad latensreduktion håller inte längre detta antagande, vilket skulle kunna göra att grafikdrivrutinens justeringar av grafikkortets klockhastighet blir direkt opassande och orsakar en negativ prestandapåverkan. Detta är i kontrast mot processorn, vars klockhastighet går att styra programmatiskt genom att ställa in energischemat via Windows-APIn (Microsoft 2021b). Programmatisk inställning av energischemat är inte del av implementationen, men som nämnt i del 3.1.4 användes energischemat ”Hög prestanda” under experimentet. Anledningen till detta är att det ska representera hur tekniken presterar när programmatisk inställning av energischemat väl har implementerats, eftersom det är möjligt. Det går att justera grafikkortets energischema i drivrutinens kontrollpanel men inte programmatiskt, så därför används standardinställningarna för grafikdrivrutinen.

5 Utvärdering

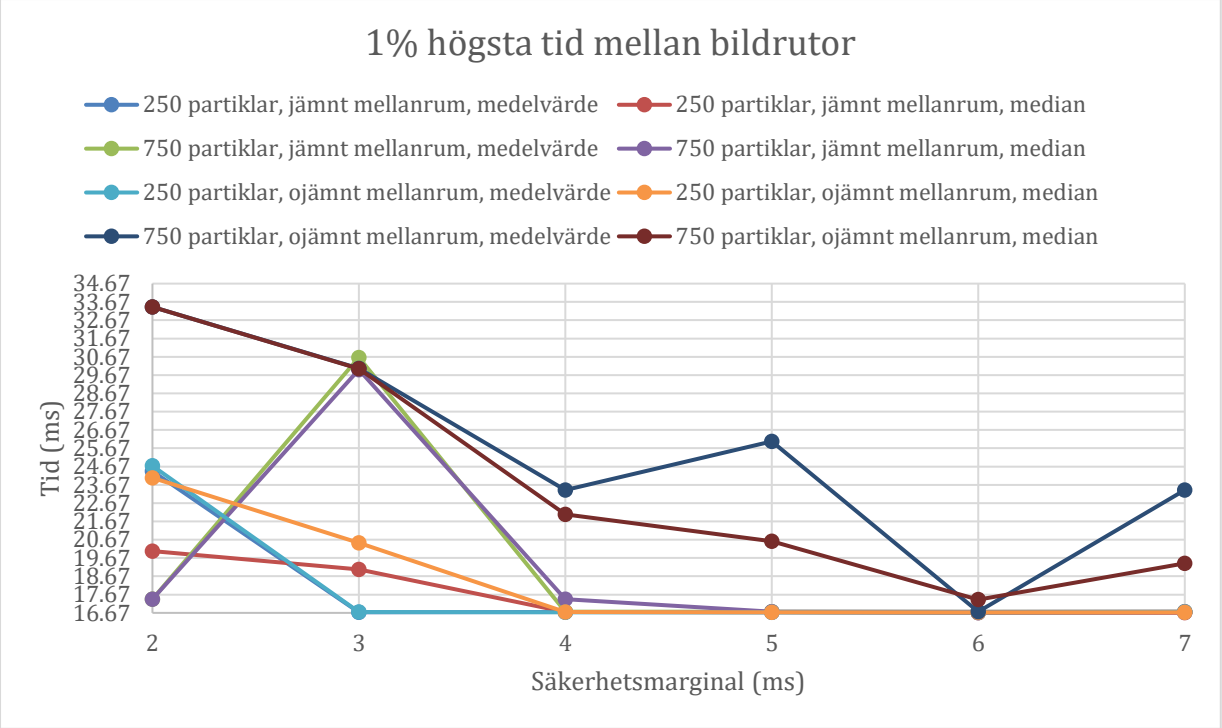
5.1 Resultat

Det fullständiga resultatet av experimentet består av 960 värden uppdelade i 32 tabeller. För läsbarhetens skull kommer endast utvalda data som är relevant för analysen i del 5.2 att presenteras i denna del, men notera att de var det fullständiga resultatet som analyserades. Det fullständiga resultatet finns i Appendix A.

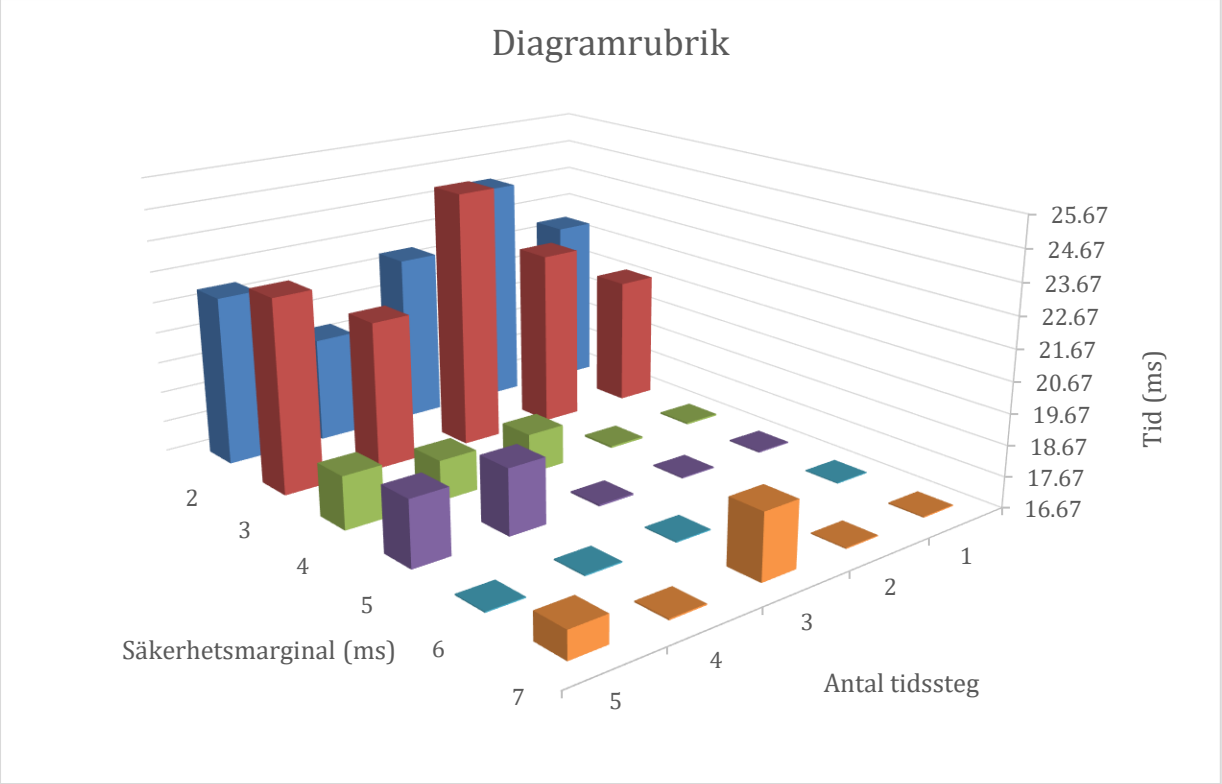
Utan latensreduktion körde alla belastningar med ett stabilt intervall mellan bildrutor på 16.7 ms utan att missa ett enda vertikalt blankningsintervall, samt en stabil latens på 49.7 ms. Testresultaten med latensreduktion presenteras i nedanstående figurer. Genomsnittlig latens ges i figur 6. Resultat för prestanda i form av 1% högsta tidmätningarna mellan bildrutor ges i figur 7 och figur 8, ur två olika perspektiv. Figur 7 visualiserar resultaten för varje experiment baserat på säkerhetsmarginal och figur 8 visualiserar det genomsnittliga resultaten för alla experiment baserat på säkerhetsmarginal och antal tidssteg som användes i prognosen.



Figur 6 Den genomsnittliga latensen vid de olika testscenariona när latensreduktion användes. Värdena i diagrammet är genomsnittet av resultaten från alla parametervärden för antal tidssteg.



Figur 7 De 1% högsta mätvärdena av tid mellan bildrutor när latensreduktion användes. Värdena i diagrammet är genomsnittet av resultaten från alla parametervärden för antal tidssteg.



Figur 8 De 1% högsta mätvärdena av tid mellan bildrutor när latensreduktion användes. Värdena i diagrammet är genomsnittet av resultaten från alla testscenarior och medelvärde/median.

5.2 Analys

Att testbelastningarna är helt fria från lagg när latensreduktion inte används utgör en god referenspunkt – för att latensreduceringen ska ha motsvarande stabilitet ska inte ett enda vertikalt blankningsintervall missas, samtidigt som den uppnår en latens som är lägre än 49.7 ms.

Latensen ökar vid tyngre belastningar och högre marginaler men skiljer sig i övrigt inte markant baserat på andra parametrar. Under testscenario 1–3, alltså vid låg och/eller stabil belastning, är prestandan stabil även vid relativt låga marginaler. Vid lätta eller stabila belastningar verkar det inte spela någon större roll om medelvärdet eller medianen av prestandavärdena används, men i testscenario 4 ger medianen bättre stabilitet vid 4–5 ms i säkerhetsmarginal. Dock kräver denna typ av belastning betydligt högre säkerhetsmarginal än de lättare och stabilare belastningarna för att uppnå hög stabilitet. Det är oklart hur många tidssteg som ska användas i prognosberäkningen för bästa resultat.

Utöver detta noterades att mätvärdena vid 3 ms säkerhetsmarginal under testscenario 2, alltså 750 partiklar med jämna mellanrum, stod ut. De hade sämre stabilitet än både lägre och högre säkerhetsmarginaler.

5.3 Slutsatser

Det mesta av den data som samlades in från experimentet stämmer överens med vad som väntades enligt hypotesen. Medianen hade i teorin bättre återhämtningsförmåga vid detta användningsområde eftersom enskilda värden som står ut filtreras bort, och detta bekräftades av resultatet. Även vid höga säkerhetsmarginaler kan en betydande latensminskning uppnås. Dock är det uppenbart att tekniken inte kan göra lika stor nytta vid instabila belastningar samt att säkerhetsmarginalen behöver justeras manuellt baserat på applikationen som tekniken implementeras i.

Det utstående mätvärdet i testscenario 2 skulle kunna tyda på en brist i latensreduceringsberäkningen. I så fall behövs antagligen mer information om renderingstrådens nuvarande stadie än enbart storleken på bildrutekön för att göra en mer korrekt estimering av beräkningstid och motverka detta edge case. Det skulle också kunna vara en konsekvens av att grafikdrivrutinen justerar grafikkortets klockhastighet på ett opassande sätt (se del 4.4).

För att svara på frågorna i problemformuleringen ger tekniken en uppenbar förbättring i latens. Så länge applikationen är relativt stabil och rätt parametervärden för prognosen väljs går det att uppnå stabilitet som är jämförbar med om latensreducering inte använts alls. Det ger bättre stabilitet att använda medianen av ett antal tidigare prestandamätningar än medelvärdet, men säkerhetsmarginalen behöver justeras efter varje applikation. Hur många tidigare prestandamätningar som ska användas för bästa resultat går dock inte att svara på utifrån resultatet.

6 Avslutande diskussion

6.1 Sammanfattning

Målet med arbetet var att implementera JIT-baserad latensreduktion för dubbelbuffrad vertikal synkronisering och utvärdera denna teknik genom experiment. Alla komponenter som krävdes för att utföra latensreduktion implementerades i ett fristående bibliotek vid namn KaJit, som i sin tur implementerades i den egenskrivna spelmotorn Kamera Engine. Kamera Engine användes sedan för att konstruera prestandatestapplikationen. Det konstruerades ett internt latensmätningsverktyg i spelmotorn som mäter tiden från det att inmatning hämtas på huvudtråden till det att buffertbytet sker på grafikdrivrutinens renderingstråd för att utvärdera tekniken.

Frågeställningen i problemformuleringen bestod av dessa frågor: Är det praktiskt och gynnsamt att använda JIT-baserad latensreduktion i spel med dubbelbuffrad vertikal synkronisering? Hur ska prognosen för framtida beräkningstid tas fram för att ge en bra avvägning mellan latens och stabilitet vid olika belastningar? Svaret på den första frågan är ja, så länge implementationen optimeras utefter applikationens prestanda. Svaret på den andra frågan är att medianen av ett antal tidigare mätningar ger bättre stabilitet än medelvärdet och att marginalen beror på applikationen.

6.2 Diskussion

6.2.1 Samhälleliga och etiska aspekter

Eftersom denna teknik minskar latens i mjukvara krävs ingen ytterligare hårdvara för att utnyttja den. Om tekniken implementeras väl i ett spel som kan dra nytta av det, exempelvis ett rytmspel, ökas tillgängligheten för att ha en högkvalitativ spelupplevelse med låg latens i det spelet. Framför allt är detta relevant vad gäller ekonomisk tillgänglighet för de som inte har råd att exempelvis köpa en ny skärm med hög och/eller dynamisk uppdateringsfrekvens. Utöver det, I kölvattnet av halvledarbristen som uppstod under 2020 och än idag påverkar tillgängligheten av viss elektronik, är det bevisligen inte längre att ta förgivet att ny hårdvara finns tillgänglig att köpa över huvud taget.

Genom att möjliggöra spelupplevelser med låg latens utan att rendera fler bildrutor än vad som kan visas för användaren finns det god potential för tekniken att minska energiförbrukningen av att spela dataspel. Detta är en ytterst relevant och viktig aspekt idag. Med sammanhanget av klimatkrisen är det önskvärt att minska energiförbrukningen för att minska klimatavtrycket av energiproduktion. Med sammanhanget av de ökande energipriserna i stora delar av Europa är det även viktigt idag att minska energiförbrukningen ur en ekonomisk synpunkt, vilket leder tillbaka till den ekonomiska tillgängligheten som tekniken kan bidra till. Dock hade den exakta energiförbrukningen jämfört med att inte använda denna teknik behövt undersökas för att komma fram till exakt hur energieffektivt JIT-baserad latensreduktion är. Som diskuterat i del 4.4 kan det vara relevant att hålla klockfrekvenserna på både CPU och GPU höga trots lägre utnyttjningsgrad och den exakta påverkan på energieffektivitet blir därmed mer komplex än att energiförbrukning skulle ha en direkt korrelation med antal bildrutor som beräknas per sekund.

Sammantaget finns god potential att kunna utnyttja denna teknik för att förbättra spelupplevelser och minska energiförbrukning utan extra kostnad för användaren, med alla förmåner som detta medför ur dagsaktuella synpunkter.

6.2.2 API-stöd och dokumentation

Något som blev uppenbart när JIT-baserad latensreduktion implementerades var att de APIer, drivrutiner och operativsystem som finns tillgängliga idag inte är designade för att kunna stödja denna teknik väl. Det hade exempelvis varit av nytta att ha ett mer gediget sätt att ta reda på när det vertikala blankningsintervallet skedde senast eller att ha tillgång till schemaläggningsbaserad blockering med ännu högre precision än 2 ms i Windows. Det hade även varit önskvärt att ha tillgång till information om skärmar med dynamisk uppdateringsfrekvens, eftersom dynamisk uppdateringsfrekvens i stort sett hade kunnat agera som dispens på deadlinen, vilket hade lättat kraven på prognosens exakthet för dessa skärmar.

Ett annat problem som uppstod under arbetets gång var att det var svårt att hitta information om de mjukvarukomponenter som användes. Det stod ingenting i Microsofts egen dokumentation om att schemaläggarens precision behöver ställas om manuellt för att uppnå högsta möjliga precision med waitable timers utan den informationen fanns enbart i ett obskyrt blogginlägg. Det var inte heller uppenbart att grafikdrivrutinen redan utför OpenGL API-anrop asynkront genom en separat renderingstråd i den mån det är möjligt, inklusive buffertbyte. Det ledde till att mycket tid slösades på att implementera en egen renderingstråd och system för att hantera asynkrona kommandon. Det upptäcktes sent in i arbetet att detta var en dubbelimplementation.

6.3 Framtida arbete

Något som hade varit intressant att undersöka vidare efter detta arbete är hur prognosen tas fram. Eftersom prognosen är en så pass viktig aspekt för att latensreduceringen ska fungera väl är det ett område som mycket väl hade kunnat gynnas av att utnyttja maskinlärning. Det hade även varit intressant att utföra energimätningar på tekniken för att få hårddata på energieffektivitetsaspekten, samt att se hur tekniken presterar vid hög CPU-belastning.

Vidare hade det varit till fördel för framtida utveckling av denna teknik att ha bättre hårdvaru- och mjukvarustöd. Som tidigare nämnt hade det varit till stor nytta om tiden för det senaste vertikala blankningsintervallet och information om skärmar med dynamisk uppdateringsfrekvens hade gjorts mer lättillgängligt via grafikdrivrutinen, samt om det gick att använda schemaläggningsbaserad blockering med en högre precision. Det hade även varit intressant att undersöka implementation av denna teknik på andra plattformar än Windows.

Slutmålet med JIT-baserad latensreducering för dubbelbuffrad vertikal synkronisering anser jag vara när den når punkten att den är generellt applicerbar och väl stödd, dels bakom kulisserna, dels bland populära spel och spelmotorer. Det är då tekniken hade gjort som störst nytta. All forskning som styr utvecklingen av tekniken åt det hållet ser jag som önskvärd.

Referenser

ARM (2011). *Mali GPU Application Optimization Guide*. Tillgänglig på Internet: https://documentation-service.arm.com/static/5ea828c99931941038df1ba8 [16-02-2022]

Banton, C. (2021). Just-in-Time (JIT). *Investopedia Dictionary*. Tillgänglig på Internet: https://www.investopedia.com/terms/j/jit.asp [17-02-2022]

Barrett, S. (2017). *stb* [Programvara]. Tillgänglig på Internet: https://github.com/nothings/stb [13-04-2022]

Battle(non)sense (2020). *NVIDIA Reflex Low Latency - How It Works & Why You Want To Use It* [Video]. Tillgänglig på Internet: https://youtu.be/QzmoLJwS6eQ?t=536 [18-02-2022]

Beder, J. (2015). *yaml-cpp* [Programvara]. Tillgänglig på Internet: https://github.com/jbeder/yaml-cpp [13-04-2022]

Boudaoud, B., Knowles, P., Kim, J. & Spjut, J. (2021). Gaming at Warp Speed: Improving Aiming with Late Warp. *Nvidia Research*. Tillgänglig på Internet: https://research.nvidia.com/publication/2021-08_Gaming-at-Warp [18-02-2022]

Claypool, M., Cockburn, A. & Gutwin, C. (2020). The Impact of Motion and Delay on Selecting Game Targets with a Mouse. *ACM Trans. Multimedia Comput. Commun. Appl.*, 16(2), Artikel 73. doi:10.1145/3390464

Cornut, O. (2022). Dear ImGui [Programvara]. Tillgänglig på Internet: https://github.com/ocornut/imgui [13-04-2022]

Dawson, B. (2020). Windows Timer Resolution: The Great Rule Change. *Random ASCII*. Tillgänglig på Internet: https://randomascii.wordpress.com/2020/10/04/windows-timer-resolution-the-great-rule-change/ [13-04-2022]

Digital Foundry (2018). *Tech Focus - V-Sync: What Is It - And Should You Use It?* [Video]. Tillgänglig på Internet: https://youtu.be/seyAzw9zEoY [16-02-2022]

Embedded Wizard (u.å.). Platform Integration Aspects: Framebuffer Concepts. *Embedded Wizard Documentation*. Tillgänglig på Internet: https://doc.embedded-wizard.de/framebuffer-concepts?v=11.00 [17-02-2022]

Epic Games (2017). *Fortnite* (19.30) [Programvara]. Tillgänglig på Internet: https://www.epicgames.com/fortnite/en-US/home [18-02-2022]

Farncomb, J. (2021). *What is a MUX Switch for Gaming Laptops?*. Tillgänglig på Internet: https://jarrods.tech/what-is-a-mux-switch-for-gaming-laptops/ [11-06-2022]

Fujita, S. (2019). *tinyobjloader* [Programvara]. Tillgänglig på Internet: https://github.com/tinyobjloader/tinyobjloader [13-04-2022]

Geelnard, M. & Löwy, C. (2021). *GLFW* (3.3.5) [Programvara]. Tillgänglig på Internet: https://www.glfw.org/ [13-04-2022]

Herberth, D. (2021). GLAD [Programvara]. Tillgänglig på Internet: https://github.com/Dav1dde/glad [13-04-2022]

ISO/IEC 14882:2014. *Programming languages – C++*. Genève: ISO/IEC

Kim, J., Knowles, P., Spjut, J., Boudaoud, B. & McGuire, M. (2020). Post-Render Warp with Late Input Sampling Improves Aiming Under High Latency Conditions. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2), Artikel 12. doi:10.1145/3406187

Korpela, P. (2014). Energy Efficiency in GPU Applications, Part 1. *ARM Community Blogs*. Tillgänglig på Internet: https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/energy-efficiency-in-gpu-applications-part-1 [16-02-2022]

Liu, S., Claypool, M., Kuwahara, A., Scovell, J. & Sherman, J. (2021). Lower is Better? The Effects of Local Latencies on Competitive First-Person Shooter Game Players. *CHI Conference on Human Factors in Computing Systems (CHI '21)*, Artikel 326. doi:10.1145/3411764.3445245

McGuire, M. (2017). *Computer Graphics Archive*. Tillgänglig på Internet: https://casual-effects.com/data/ [24-05-2022]

Microsoft (2015). *Windows 10 Home* (21H2, 19044.1706, Windows Feature Experience Pack 120.2212.4170.0) [Programvara].

Microsoft (2021a). DXGI flip model. *Microsoft Docs: Windows app development documentation*. Tillgänglig på Internet: https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/dxgi-flip-model [16-02-2022]

Microsoft (2021b). PowerSetActiveScheme function (powersetting.h). *Microsoft Docs: Windows app development documentation*. Tillgänglig på Internet: https://docs.microsoft.com/en-us/windows/win32/api/powersetting/nf-powersetting-powersetactivescheme [10-06-2022]

Nola, R., Sankey, H. (2007/2014). *Theories of Scientific Method: An Introduction*, 1st Edition. New York: Routledge.

Nvidia Corporation (u.å.). Publications. *Nvidia Research*. Tillgänglig på Internet: https://research.nvidia.com/publications [18-02-2022]

Nvidia Corporation (2022). *Nvidia Game Ready Driver* (512.77) [Programvara].

OpenGL Wiki contributors (2019). Query Objects. *OpenGL Wiki*. Tillgänglig på Internet: https://www.khronos.org/opengl/wiki/Query-Object [25-05-2022]

OpenGL Wiki contributors (2021). Synchronization. *OpenGL Wiki*. Tillgänglig på Internet: https://www.khronos.org/opengl/wiki/Synchronization [25-05-2022]

Oracle (u.å.). Double Buffering and Page Flipping. *The Java™ Tutorials*. Tillgänglig på Internet: https://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html [16-02-2022]

PCMag (u.å.). Definition of latency. *PCMag Encyclopedia*. Tillgänglig på Internet: https://www.pcmag.com/encyclopedia/term/latency [17-02-2022]

Russinovich. M. & Solomon, D. (2009). *Windows Internals*, 5th Edition. Redmond: Microsoft Press.

Schneider, S. (2020). Introducing NVIDIA Reflex: Optimize and Measure Latency in Competitive Games. *Geforce News*. Tillgänglig på Internet: https://www.nvidia.com/en-us/geforce/news/reflex-low-latency-platform/ [16-02-2022]

Segal, M. & Akeley, K. (2013). *The OpenGL® Graphics System: A Specification*. Beaverton: The Khronos Group Inc.. Tillgänglig på Internet: https://www.khronos.org/registry/OpenGL/specs/gl/glspec43.core.pdf [13-04-2022]

Silberschatz, A., Galvin, P.B. & Gagne, G. (2008). *Operating System Concepts,* 8th Edition. New Jersey: John Wiley & Sons, Inc..

Spjut, J., Boudaoud, B., Binaee, K., Kim, J., Majercik, A., McGuire, M., Luebke, D. & Kim, J. (2019). Latency of 30 ms Benefits First Person Targeting Tasks More Than Refresh Rate Above 60 Hz. *SA '19: SIGGRAPH Asia 2019 Technical Briefs*, ss. 110-113. doi:10.1145/3355088.3365170

Spjut, J., Boudaoud, B. & Kim, J. (2021, preprint). A Case Study of First Person Aiming at Low Latency for Esports. *EHPHCI 2021: ACM CHI Workshop on Esports and High Performance Human Computer Interaction*. doi:10.31219/osf.io/nu9p3

Svenska Akademien (1997). Submultipel. *Svenska Akademiens ordbok*. Lund: Svenska Akademien. Tillgänglig på Internet: https://www.saob.se [18-02-2022]

Svenska Akademien (2015). *Svenska Akademiens ordlista över svenska språket*. Stockholm: Nordstedts.

The kernel development community (u.å.). Kernel Mode Setting (KMS). *The Linux Kernel documentation*, 5.17.0-rc4. Tillgänglig på Internet: https://docs.kernel.org/gpu/drm-kms.html#vertical-blanking [16-02-2022]

Wasson, S. (2011). *Inside the second: A new look at game benchmarking*. Tillgänglig på Internet: https://techreport.com/review/21516/inside-the-second-a-new-look-at-game-benchmarking/ [9-06-2022]

Žilys, T. (2020). Fixing Time.deltaTime in Unity 2020.2 for smoother gameplay: What did it take?. *Unity Blog*. Tillgänglig på Internet: https://blog.unity.com/technology/fixing-time-deltatime-in-unity-2020-2-for-smoother-gameplay-what-did-it-take [16-02-2022]

Appendix A - Mätvärden från experiment

Utan latensreduktion

Utan latensreduktion körde alla belastningar med ett stabilt intervall mellan bildrutor på 16.7 ms utan att missa ett enda vertikalt blankningsintervall, samt en stabil latens på 49.7 ms. Resultaten för alla testbelastningar med latensreduktion presenteras i nedanstående tabeller.

Testscenario 1: genomsnitt 250 partiklar med jämna mellanrum

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medelvärdet av prestandadatan.

Tabell 1 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017135	0,01666	0,016667	0,016667	0,016667	0,016667
2 tidssteg	0,01686	0,016667	0,016667	0,016667	0,016667	0,016667
3 tidssteg	0,016667	0,016667	0,016667	0,016667	0,016667	0,016667
4 tidssteg	0,016667	0,016667	0,016667	0,016667	0,016667	0,016667
5 tidssteg	0,016667	0,016667	0,016667	0,016667	0,016667	0,016667

Tabell 2 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,021387	0,016721	0,016724	0,016672	0,01667	0,016673
2 tidssteg	0,018861	0,016673	0,016673	0,016671	0,016671	0,016671
3 tidssteg	0,016672	0,016674	0,016675	0,016676	0,01667	0,016673
4 tidssteg	0,016672	0,01667	0,016682	0,016675	0,016671	0,016672
5 tidssteg	0,016679	0,016672	0,016673	0,016673	0,016672	0,016672

Tabell 3 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,033387	0,016753	0,01677	0,016692	0,016672	0,016701

2 tidssteg	0,038571	0,0167	0,016694	0,016676	0,016677	0,016681
3 tidssteg	0,016694	0,0167	0,016723	0,016717	0,016673	0,016695
4 tidssteg	0,016687	0,016676	0,016748	0,016717	0,016678	0,016695
5 tidssteg	0,016742	0,01668	0,016699	0,016696	0,01669	0,016695

Tabell 4 Medelvärde av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,010555	0,01103	0,012131	0,013261	0,014253	0,015198
2 tidssteg	0,01073	0,011202	0,012486	0,01316	0,015127	0,015236
3 tidssteg	0,01024	0,011223	0,012227	0,013202	0,0142	0,01524
4 tidssteg	0,010285	0,011232	0,01224	0,013276	0,014263	0,01521
5 tidssteg	0,010218	0,011188	0,012229	0,013227	0,014287	0,015254

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medianen av prestandadatan.

Tabell 5 Median av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017001	0,016667	0,016667	0,016668	0,016667	0,016667
2 tidssteg	0,016667	0,016734	0,016667	0,016667	0,016667	0,016667
3 tidssteg	0,016667	0,016667	0,016667	0,01666	0,016667	0,016667
4 tidssteg	0,016667	0,016667	0,016667	0,016667	0,016667	0,016668
5 tidssteg	0,016667	0,016667	0,016668	0,016667	0,016667	0,016667

Tabell 6 Median av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,020053	0,016707	0,016735	0,016694	0,016675	0,016671
2 tidssteg	0,016673	0,01767	0,016672	0,016678	0,01667	0,016671
3 tidssteg	0,016676	0,016848	0,016676	0,01667	0,016672	0,016671
4 tidssteg	0,016674	0,01667	0,016673	0,016674	0,016671	0,016672
5 tidssteg	0,016676	0,016677	0,016677	0,016675	0,01667	0,01667

Tabell 7 Median av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,033373	0,016747	0,016775	0,016732	0,016714	0,016675
2 tidssteg	0,01669	0,02669	0,016689	0,016726	0,016672	0,016678
3 tidssteg	0,016729	0,01841	0,016724	0,016702	0,016686	0,016681
4 tidssteg	0,016711	0,016673	0,0167	0,016707	0,016678	0,016686
5 tidssteg	0,01673	0,016722	0,01672	0,016717	0,016686	0,016671

Tabell 8 Median av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,010626	0,01114	0,01224	0,01315	0,014186	0,015254
2 tidssteg	0,010277	0,011411	0,012172	0,013186	0,014261	0,015192
3 tidssteg	0,010267	0,011264	0,012194	0,01324	0,014279	0,015246
4 tidssteg	0,01023	0,011239	0,012222	0,013267	0,014214	0,015229
5 tidssteg	0,010219	0,011294	0,012232	0,01325	0,014253	0,015258

Testscenario 2: genomsnitt 750 partiklar med jämna mellanrum

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medelvärdet av prestandadatan.

Tabell 9 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,016701	0,016767	0,016667	0,016667	0,01666	0,016667
2 tidssteg	0,016667	0,016867	0,01666	0,016667	0,016667	0,016667
3 tidssteg	0,016667	0,016934	0,01666	0,01666	0,016667	0,016667
4 tidssteg	0,016667	0,016767	0,016667	0,01666	0,016667	0,016667
5 tidssteg	0,016667	0,017001	0,01666	0,016667	0,016667	0,01666

Tabell 10 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017054	0,017690	0,016719	0,016704	0,016701	0,016701
2 tidssteg	0,016718	0,018699	0,016721	0,016697	0,016699	0,016704
3 tidssteg	0,016715	0,01936	0,016725	0,016702	0,01669	0,016697
4 tidssteg	0,016714	0,017704	0,016716	0,016697	0,016704	0,016699
5 tidssteg	0,01672	0,020026	0,016721	0,016698	0,016696	0,016695

Tabell 11 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,020111	0,026688	0,01673	0,016737	0,016737	0,016734
2 tidssteg	0,016743	0,03327	0,016741	0,016727	0,016729	0,01676
3 tidssteg	0,01674	0,03327	0,01674	0,016738	0,01671	0,016726
4 tidssteg	0,016731	0,026673	0,016738	0,016725	0,016741	0,01673
5 tidssteg	0,016755	0,03329	0,016735	0,01672	0,016727	0,01672

Tabell 12 Medelvärde av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,016721	0,017755	0,018843	0,019881	0,020875	0,0219
2 tidssteg	0,016826	0,017648	0,018837	0,01991	0,020868	0,021891
3 tidssteg	0,016802	0,017664	0,018839	0,019893	0,020864	0,02187
4 tidssteg	0,016833	0,01768	0,018827	0,019903	0,020898	0,021867
5 tidssteg	0,016806	0,017701	0,018823	0,01987	0,020879	0,021897

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medianen av prestandadatan.

Tabell 13 Median av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**

1 tidssteg	0,016701	0,016667	0,016701	0,016667	0,01666	0,01666
2 tidssteg	0,016667	0,017034	0,01666	0,016667	0,01666	0,016667
3 tidssteg	0,016667	0,016901	0,016667	0,01666	0,016667	0,016667
4 tidssteg	0,016667	0,016934	0,01666	0,01666	0,01666	0,01666
5 tidssteg	0,016667	0,016834	0,016667	0,01666	0,016667	0,01666

Tabell 14 Median av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017055	0,016692	0,017055	0,016714	0,016704	0,016696
2 tidssteg	0,016712	0,020359	0,0167	0,016696	0,016696	0,016697
3 tidssteg	0,016713	0,019031	0,01672	0,016697	0,016696	0,016701
4 tidssteg	0,016713	0,019365	0,016719	0,01669	0,01669	0,016698
5 tidssteg	0,016709	0,018364	0,016717	0,016697	0,016698	0,016702

Tabell 15 Median av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,0201	0,016723	0,02006	0,016758	0,01673	0,016725
2 tidssteg	0,016732	0,033278	0,016735	0,016723	0,016722	0,016724
3 tidssteg	0,016763	0,033277	0,016739	0,016721	0,016733	0,016757
4 tidssteg	0,016738	0,033305	0,016759	0,016724	0,016729	0,016744
5 tidssteg	0,016731	0,033264	0,016756	0,01672	0,016729	0,016731

Tabell 16 Median av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,016691	0,017676	0,018876	0,019881	0,020868	0,021914
2 tidssteg	0,016819	0,017691	0,018831	0,019895	0,020866	0,021902
3 tidssteg	0,016803	0,017667	0,018805	0,01988	0,020886	0,021885
4 tidssteg	0,0168	0,017718	0,01882	0,019903	0,02088	0,021908
5 tidssteg	0,016811	0,017647	0,018833	0,019884	0,020889	0,021926

Testscenario 3: genomsnitt 250 partiklar med slumpade intervaller

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medelvärdet av prestandadatan.

Tabell 17 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017168	0,016667	0,016668	0,016667	0,016667	0,016667
2 tidssteg	0,016701	0,016667	0,016667	0,016667	0,016667	0,016667
3 tidssteg	0,016767	0,016667	0,016667	0,016667	0,016667	0,016667
4 tidssteg	0,016667	0,016667	0,016667	0,016667	0,016667	0,016667
5 tidssteg	0,016767	0,016667	0,016667	0,016667	0,016667	0,016667

Tabell 18 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,021695	0,01672	0,016704	0,016676	0,016674	0,016673
2 tidssteg	0,017023	0,016672	0,01667	0,016671	0,016673	0,016677
3 tidssteg	0,017712	0,016673	0,016674	0,016674	0,016671	0,016672
4 tidssteg	0,016697	0,016673	0,016672	0,016673	0,01667	0,016676
5 tidssteg	0,017699	0,016697	0,016673	0,016672	0,016673	0,016693

Tabell 19 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,033353	0,016763	0,016744	0,016728	0,01671	0,016695
2 tidssteg	0,020056	0,016686	0,01669	0,016679	0,016702	0,016716
3 tidssteg	0,026693	0,016696	0,016712	0,016712	0,01668	0,016688
4 tidssteg	0,016745	0,016698	0,016685	0,016699	0,016675	0,016719
5 tidssteg	0,02669	0,016712	0,0167	0,016691	0,016698	0,016742

Tabell 20 Medelvärde av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,010447	0,011023	0,011893	0,013166	0,014205	0,01502
2 tidssteg	0,01006	0,011257	0,012281	0,013138	0,014184	0,01512
3 tidssteg	0,011181	0,011095	0,012168	0,013101	0,014178	0,0151
4 tidssteg	0,01017	0,01119	0,012185	0,013145	0,014136	0,015046
5 tidssteg	0,010299	0,010994	0,012103	0,013202	0,014038	0,016028

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medianen av prestandadatan.

Tabell 21 Median av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017567	0,016701	0,016667	0,016667	0,016667	0,016667
2 tidssteg	0,016801	0,016667	0,016667	0,016667	0,016667	0,016667
3 tidssteg	0,016734	0,016667	0,016667	0,016667	0,016667	0,016667
4 tidssteg	0,016667	0,016667	0,016667	0,016667	0,016667	0,016667
5 tidssteg	0,016667	0,016801	0,016667	0,016667	0,016667	0,016667

Tabell 22 Median av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,025694	0,017052	0,016728	0,016676	0,01667	0,016684
2 tidssteg	0,018037	0,016673	0,01667	0,016674	0,016671	0,016671
3 tidssteg	0,017362	0,016677	0,01667	0,016673	0,01667	0,01667
4 tidssteg	0,016695	0,016675	0,016673	0,016673	0,016674	0,016673
5 tidssteg	0,016698	0,018256	0,016674	0,016672	0,016674	0,016671

Tabell 23 Median av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**

1 tidssteg	0,03338	0,020079	0,01677	0,016726	0,016672	0,016732
2 tidssteg	0,030043	0,016703	0,016672	0,016706	0,016681	0,01668
3 tidssteg	0,023374	0,01673	0,016696	0,016696	0,016705	0,01669
4 tidssteg	0,016736	0,01672	0,016696	0,016695	0,016707	0,016699
5 tidssteg	0,016735	0,0323	0,016715	0,01669	0,01671	0,016684

Tabell 24 Median av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,011041	0,011022	0,012062	0,012917	0,01419	0,015699
2 tidssteg	0,01034	0,011131	0,012157	0,013073	0,014192	0,015159
3 tidssteg	0,010275	0,011028	0,01199	0,01315	0,014143	0,015079
4 tidssteg	0,011039	0,011128	0,012218	0,013169	0,014070	0,014958
5 tidssteg	0,010217	0,011475	0,012251	0,013178	0,014198	0,015165

Testscenario 4: genomsnitt 750 partiklar med slumpade intervaller

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medelvärdet av prestandadatan.

Tabell 25 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017168	0,016801	0,01676	0,01666	0,01666	0,016834
2 tidssteg	0,017334	0,016701	0,016734	0,017001	0,016667	0,01666
3 tidssteg	0,017167	0,017034	0,016801	0,016867	0,016667	0,016734
4 tidssteg	0,017134	0,017067	0,016667	0,016667	0,016667	0,016667
5 tidssteg	0,017167	0,017034	0,016701	0,016801	0,016667	0,01676

Tabell 26 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,021724	0,018048	0,01773	0,016710	0,0167	0,018371
2 tidssteg	0,023393	0,017052	0,017396	0,020036	0,016695	0,016697

3 tidssteg	0,021726	0,020377	0,018056	0,018705	0,016698	0,01736
4 tidssteg	0,021392	0,020715	0,01672	0,016706	0,016705	0,0167
5 tidssteg	0,021727	0,02037	0,017065	0,018034	0,016709	0,017696

Tabell 27 Medelvärde av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,033377	0,030003	0,026705	0,016751	0,016735	0,033335
2 tidssteg	0,033404	0,020081	0,02337	0,033349	0,016733	0,01672
3 tidssteg	0,033403	0,033355	0,030008	0,033328	0,016728	0,023378
4 tidssteg	0,0334	0,033362	0,016757	0,016738	0,016743	0,016749
5 tidssteg	0,0334	0,03334	0,0201	0,030032	0,01674	0,026701

Tabell 28 Medelvärde av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,01756	0,017842	0,019054	0,019886	0,020893	0,02235
2 tidssteg	0,01809	0,017867	0,018738	0,020647	0,0208	0,021897
3 tidssteg	0,017832	0,018467	0,01915	0,020604	0,020908	0,022033
4 tidssteg	0,017483	0,018395	0,018881	0,0199	0,020856	0,021896
5 tidssteg	0,017678	0,018455	0,018822	0,020231	0,020881	0,022125

Nedan presenteras resultaten för detta testscenario när latensreduktionen använde medianen av prestandadatan.

Tabell 29 Median av prestandadata, genomsnittlig tid mellan bildrutor

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017301	0,016767	0,016734	0,016701	0,016667	0,016667
2 tidssteg	0,017101	0,016834	0,01666	0,016701	0,01666	0,016667
3 tidssteg	0,017234	0,0169	0,016767	0,016701	0,016667	0,016701
4 tidssteg	0,017134	0,016834	0,016701	0,016667	0,016667	0,016767
5 tidssteg	0,017367	0,016734	0,016734	0,016767	0,016701	0,016667

Tabell 30 Median av prestandadata, genomsnittlig tid mellan bildrutor (10% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,023053	0,017708	0,017396	0,017035	0,016712	0,016694
2 tidssteg	0,021065	0,018375	0,016729	0,01704	0,016696	0,016694
3 tidssteg	0,022385	0,019039	0,017726	0,017037	0,016705	0,017029
4 tidssteg	0,021397	0,018376	0,017065	0,016703	0,016694	0,01769
5 tidssteg	0,023729	0,01737	0,017393	0,0177	0,017025	0,016688

Tabell 31 Median av prestandadata, genomsnittlig tid mellan bildrutor (1% högsta värden)

Tid mellan bildrutor (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,033372	0,026690	0,023394	0,020053	0,01674	0,016727
2 tidssteg	0,033409	0,03331	0,016754	0,020053	0,016729	0,016718
3 tidssteg	0,033377	0,033335	0,026683	0,020091	0,016732	0,020052
4 tidssteg	0,033402	0,03332	0,020068	0,016734	0,016735	0,02669
5 tidssteg	0,033422	0,023393	0,023387	0,026	0,020034	0,016732

Tabell 32 Median av prestandadata, genomsnittlig latens

Latens (s)	**2 ms**	**3 ms**	**4 ms**	**5 ms**	**6 ms**	**7 ms**
1 tidssteg	0,017642	0,018025	0,018989	0,019974	0,020896	0,02189
2 tidssteg	0,017512	0,018019	0,018835	0,019971	0,020869	0,021854
3 tidssteg	0,017699	0,018128	0,018837	0,019986	0,020864	0,021997
4 tidssteg	0,017455	0,018071	0,018941	0,019887	0,020893	0,022098
5 tidssteg	0,017968	0,017888	0,019053	0,020114	0,020981	0,021853