Examensarbete

HÖGSKOLAN
I SKÖVDE

HANTERING OCH OMSÄTTNING AV OBJEKT I OOD KONTRA DOD

Hur presterar OOD jämfört med DOD när många objekt måste hanteras, skapas och förstöras?

MANAGEMENT AND TURNOVER OF OBJECTS IN OOD VS. DOD

How does OOD perform compared to DOD when many objects must be managed, created, and destroyed?

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2025

Viktor Andersson

Handledare: Mikael Thieme
Examinator: Mikael Johannesson

Sammanfattning

Denna studie ämnar att undersöka hur objektorienterad design (OOD) och dataorienterad design (DOD) presterar när många objekt måste hanteras, skapas och förstöras. DOD har visats vara effektivare på att lagra data i minnet och vara lättare att parallellisera. Detta innebär att DOD kan dra mer nytta av systemets resurser, vilket leder till bättre prestanda jämfört med OOD. Ett vanligt förekommande programmeringsmönster inom OOD är object pooling (OODOP) som används för att minska effekten av allokeringar på prestandan. Istället för att allokera och avallokera objekt återanvänds de. När program allokerar objekt påverkas prestandan, detta hanteras på olika sätt för OOD och DOD. Experiment utfördes genom att skapa ett antal objekt i en scen. Varje frame omsätts objekt genom att förstöra och sedan skapa lika många nya objekt. Resultatet visar att OODOP presterar bäst medans OOD bör användas för färre antal objekt och DOD när det är många objekt.

Nyckelord: Unity, DOTS, Cache, Allokering, Objektorienterad design, Dataorienterad design

Innehållsförteckning

1 Introduktion

Objektorienterad design (OOD) har fokuset på objekt och hur de relaterar till varandra. För dataorienterad design (DOD) är fokuset på data, där dess användning och lagring i minnet har stor betydelse. DOD lägger upp data i minnet sekventiellt, vilket ger sammanhängande data som optimerar användandet av processorns cache. DOD är också enklare att parallellisera då data och beteende är separerat, vilket är motsatsen till OOD. Parallellisering innebär att arbete sprids ut över flera trådar i processorn. För experimenten kommer dessa paradigm användas i samband med komponentsystem. I komponentsystem byggs objekt upp genom att kombinera olika komponenter. Till exempel kan ett fiendeobjekt vara uppbyggt av en rörelse- och skjutkomponent.

Unity är en välkänd spelmotor som i grunden använder OOD men som också stödjer DOD genom deras Data-Oriented Technology Stack (DOTS). DOTS är en samling teknologier som gör det möjligt att skriva dataorienterad kod i Unity. DOTS inkluderar ett Entity Component System (ECS) men även parallelliserings- och kompileringsverktyg som används för att öka prestandan av programmet. Med DOTS måste strukturella förändringar, som att lägga till och ta bort objekt, hända på huvudtråden samt måste all parallellisering stanna medans dessa förändringar tillämpas.

Allokeringar händer när objekt skapas och förstörs vilket påverkar prestandan. Inom OOD och DOD händer allokeringar på olika sätt och har därför olika påverkan på prestandan. OOD använder en garbage collector som automatisk avallokerar oanvända objekt, detta i sig har en prestandapåverkan då garbage collector:n behöver söka igenom hela minnet. För DOD föredras datatyper som float, int och struct för att de är snabbare att allokera, men de använder ingen garbage collector och kan därav behöva manuellt avallokeras. I DOD ska data inte kunna ändras efter instansiering och ska istället bytas ut helt, därför används datatyper som är snabba att allokera. Object pooling (OODOP) är ett programmeringsmönster som används inom OOD när många objekt skapas och förstörs ofta. Genom att återanvända objekt behöver data endast allokeras en gång i början. När objekt återanvänds kan de behöva ändras på för att återspegla objektets nya tillstånd.

Experiment utfördes för OOD, DOD och OODOP genom att använda komponentsystem. Experimenten började med att skapa ett antal objekt i scenen. Varje frame omsätts objekt genom att först förstöra och sedan skapa lika många nya objekt. Medelvärdet och standardavvikelsen kunde beräknas genom att kolla hur lång tid varje frame tog att exekvera.

Resultatet av experimenten visade att OODOP presterade bäst. För OODOP utfördes ytterligare ett test för att bättre efterlikna spel som implementerar mönstret. Detta är på grund av att de föregående testerna inte representerade det väl. Testet visade att OODOP presterade sämre än det föregående testet, men lyckades fortfarande prestera bättre än OOD och DOD. Vid färre antal objekt presterade DOD sämre än OOD, medans vid högre antal objekt presterade DOD bättre.

2 Bakgrund

Syftet med arbetet är att undersöka hur objektorienterad och dataorienterad design presterar när många objekt hanteras, skapas och förstörs.

Objektorienterad design (OOD) har en del prestandaproblem jämfört med dataorienterad design (DOD). DOD fokuserar på hur data används och hur den läggs ut i minnet. Med fokuset på data, som Turpeinen (2020) påvisar i sin studie, kan DOD prestera bättre än OOD vid större mängder objekt. Unity är en spelmotor som använder OOD. Genom Data-Oriented Technology Stack (DOTS) kan även DOD användas i Unity. Skapandet och förstörandet av objekt kommer att allokera eller avallokera minne. Detta hanteras på olika sätt i OOD gentemot DOD och har olika påverkan på prestandan.

2.1 Komponentorienterad programmering

Komponentorienterad programmering fokuserar på att bygga upp objekt genom komponenter, där komposition föredras över arv. Genom olika kombinationer av komponenter kan unika objekt skapas. Användandet av komponenter ökar återanvändbarheten och underhållbarheten av programmet. Exempelvis kan både ett tåg och en bil röra på sig, men istället för att behöva implementera rörelsen för varje objekt kan istället en rörelsekomponent användas. Rörelsekomponenten kan då skrivas en gång och återanvändas. Detta har en ytterligare fördel att underhåll blir enklare då alla objekt som rör på sig använder samma beteende (Lowy 2005). Att bygga objekt med hjälp av komponenter leder också till att fler objektinstanser skapas gentemot att använda arv. Detta är en nackdel då skapandet av instanser påverkar prestandan. Komponentorienterad programmering används i komponentsystem vilket är ett sätt att skapa objekt genom att lägga till olika komponenter (Unity Technologies 2017).

2.2 Objektorienterad och Dataorienterad design

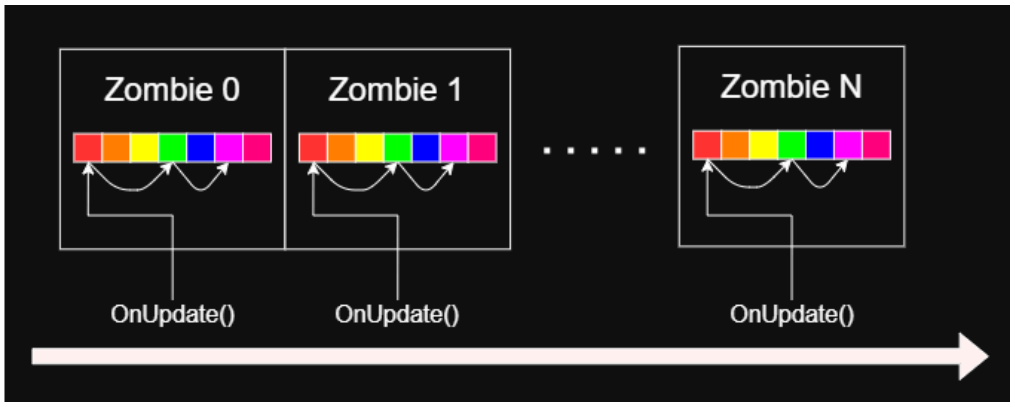
Objektorienterad design (OOD) är ett programmeringsparadigm med fokuset på objekt (klasser) som innehåller både data och beteende. Detta är ett sätt att mer likna hur vi tänker om objekt i verkliga livet. Till exempel är en bil ett objekt vilket består av andra objekt som en motor och hjul. OOD erbjuder fördelar som återanvändbarhet, modularitet och abstraktion (Aniche 2024).

Inom dataorienterad design (DOD) ligger fokuset på data, dess användning och hur den lagras i minnet. Genom att optimera upplägget av data i minnet kan bättre prestanda uppnås. Till skillnad från OOD separeras data och beteende. Data kan då lagras i en lista som sedan skickas till ett system som då utför beteendet. Eftersom att data och beteende är separerade blir det enklare att parallellisera arbetet över flera trådar vilket är en stor fördel DOD har över OOD (Llopis 2009).

Båda paradigmen har sina för- och nackdelar men fokuset för detta arbetet kommer att ligga på prestandan. För OOD är upplägget av data i minnet inte optimalt. Till exempel kan ett zombieobjekt vara uppbyggt med flera olika komponenter genom att använda komponentorienterad programmering. Komponenterna kommer att ligga i en lista på zombieobjektet.

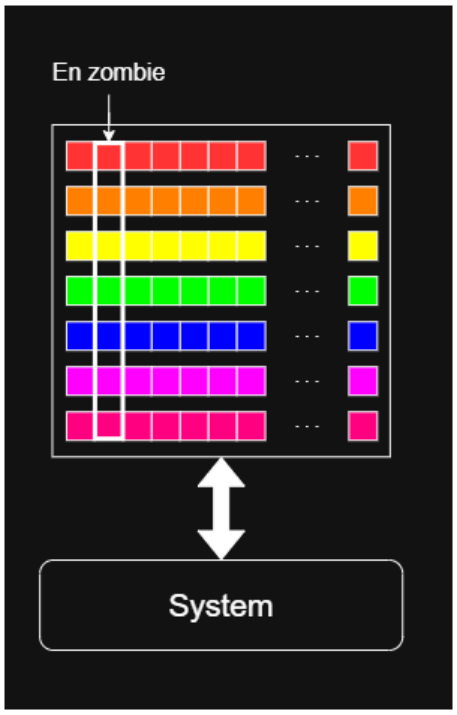
Som ett exempel är zombieobjektet en fiende som har en riktningskomponent vilket tar hand

om att rikta zombien mot ett mål. Exempelvis kan en spelare vara målet. Det finns även en hastighetskomponent som håller koll på zombiens hastigheten. Dessa två komponenter kan användas för att uppdatera positionskomponenten vilket kommer att få zombien att röra på sig mot spelaren. I OOD kan det likna figur 1.



Figur 1, visar ett vanligt förekommande problem med OOD. Mellan de komponenter som används för en beräkning (färgade objekt med pilar) hamnar andra komponenter (färgade objekt utan pilar).

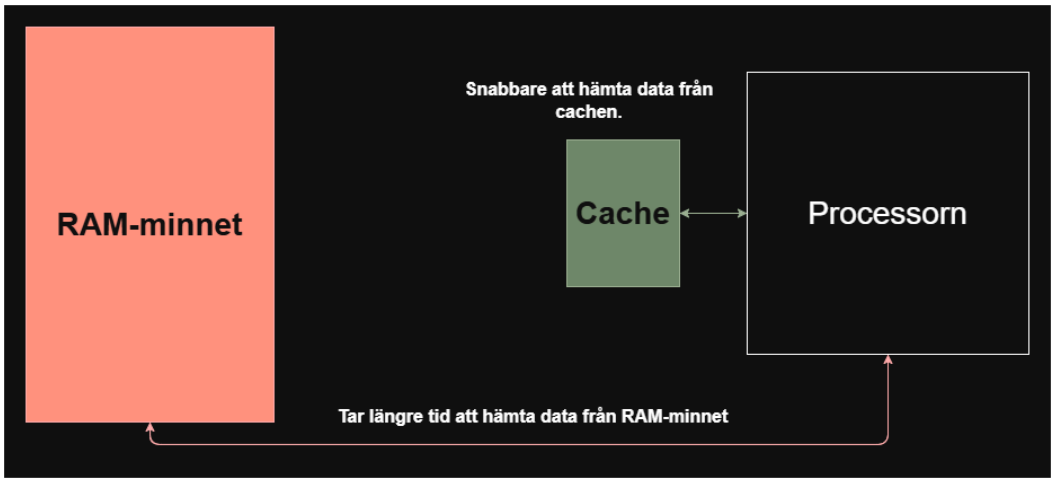
Vid en uppdatering kommer hela zombieobjektet att laddas in, vilket inkluderar alla komponenterna, även om de inte kommer användas för operationen. Som figur 1 visar blir det en massa onödig data som följer med mellan positionskomponenten (magenta), hastighetskomponenten (röd) och riktningskomponenten (grön). Figuren visar ett vanligt förekommande problem med OOD och är alltså inte sant för alla OOD-implementationer. Dessutom är upplägget av att zombieobjekten och komponenterna ligger bredvid varandra i minnet ett antagande och kan variera beroende på implementation. Detta kan innebära att mer eller mindre data finns mellan objekt och komponenter. Men för DOD som har fokuset på data kommer minnesupplägget för ett zombieobjekt likna figur 2.



Figur 2, visualiserar hur DOD lagrar zombieobjektet i minnet och hur systemet kan hämta de komponenter (färgade objekten) som behövs för operationen.

Här placeras alla komponenter av samma typ efter varandra i minnet, ett zombieobjekt blir då en kolumn av komponenterna i figur 2. Systemet kan då välja komponenterna som behövs för operationen istället för att behöva hämta hela zombieobjektet. Enligt Llopis (2009) så optimerar detta användningen av processorns minne då data helst ska vara homogen och sammanhängande. Processorn kan då jobba på data sekventiellt utan att behöva "hoppa" till nästa del av data. Llopis (2009) tar även upp hur upplägget av att separera data och beteende från varandra leder till att parallellisering enklare kan implementeras. Även Nystrom (2021) påpekar hur upplägget av minnet optimerar användandet av processorns cache.

Nystrom (2021) beskriver att genom åren har processorn blivit snabbare medans RAM-minnet inte har följt med i samma utvecklingstakt. Detta leder till att processorn måste sitta och vänta på att data ska ta sig hela vägen från RAM-minnet. För att lösa detta har processorer fått en cache. Cachen är ett litet men snabbt minne som sitter nära processorn. När processorn hämtar data från RAM-minnet hämtas inte bara den data som efterfrågas utan också data runt omkring som sedan lagras i cachen. När processorn utför arbete kollar den först efter data i cachen. Om efterfrågad data finns i cachen kan den snabbt hämtas, annars kommer en *cache miss* att inträffa. Som figur 3 visar kommer processorn behöva hämta data från RAM-minnet vilket är långsammare. Det är precis detta som DOD ämnar att förbättra. Genom att ha all data efter varandra i minnet kommer mer relevant data följa med till cachen och mindre cache misses kommer att inträffa. Gorman (2022) tar även upp att effektiv användning av cachen leder till att processorn kan arbeta igenom data snabbare. Det gör att processorn kan gå in i viloläge snabbare, och därmed förbruka mindre el.



Figur 3, visar hur RAM-minnet är längre bort vilket gör att data tar längre tid att hämta medans cachen är närmare och därför snabbare att hämta ifrån.

2.3 Unity DOTS

Unity är en spelmotor som i grunden använder OOD (Unity Technologies 2025c). Data-Oriented Technology Stack (DOTS) är en samling teknologier som möjliggör dataorienterad design i Unity. Basen för DOTS är ett Entity Component System (ECS), ett mönster som bygger på DOD. ECS-mönstret delas upp i entiteter, komponenter och system. Entiteter är ett id som refererar till komponenter och innehåller oftast ingen annan data. Komponenter är oftast bara data medans system hämtar komponenter för att utföra ett visst beteende på dem. I ECS-mönstret märks ett tydligt inflytande från DOD eftersom att data och beteende är separerade, samt ligger data kontinuerligt efter varandra i minnet (Unity

Technologies 2024, 2025a).

Unitys ECS utnyttjar parallellisering och Burst compiler för att förbättra prestandan ännu mer. Burst compiler är ett kompileringsverktyg som omvandlar bytekod till optimerad lågnivåkod. Burst Compiler gör detta genom att utföra samma operation på flera objekt samtidigt vilket förbättrar prestandan (Unity Technologies 2024).

Problemet med Unitys ECS är att strukturella förändringar, som skapandet och förstörandet av entiteter, måste hanteras på huvudtråden. När strukturella förändringar sker måste också all parallellisering stanna och vänta tills dessa förändringar har tillämpats. Detta påverkar prestandan negativt eftersom allt måste ske på huvudtråden och att inget annat arbete kan jobbas på parallellt (Unity Technologies 2024, 2025a).

2.4 Allokering och avallokering

Allokeringar händer när data skapas och behöver sparas i minnet, medan avallokeringar händer när data inte längre används och rensas från minnet. I Unity används en garbage collector som automatiskt hanterar avallokeringen av objekt som inte längre används. Detta är ett vanligt fall för OOD men det kan variera. För DOTS måste avallokeringar hanteras manuellt och använder därav inte någon garbage collector (Unity Technologies 2025b). Sharvit (2022) påpekar hur data inom DOD inte bör kunna ändras efter att den har skapats. Detta innebär att data måste bytas ut helt. Datatyper som int, float och struct är jämfört med klasser snabbare att allokera och är därför att föredra. Men, likt i DOTS, kan dessa datatyper inte använda en garbage collector och kan behöva avallokeras manuellt. Garbage collector:n har en påverkan på prestandan eftersom att den måste gå igenom minnet för att hitta objekt som inte längre används. Den är också oförutsägbar när det kommer till att avallokera minne. Detta kan leda till prestandaproblem eftersom programmet blir tvingat att vänta på garbage collector:n då stora datamängder behöver avallokeras (Microsoft 2023).

Object pooling är ett programmeringmönster som används inom OOD vilket minskar prestandapåverkan av att allokera och avallokera objekt. Detta uppnås genom att skapa alla objekt från början och sedan aktivera eller avaktivera objekten när de behövs kontra när de inte behövs längre. Processorn behöver då inte bli översvämmad med anrop om att skapa och förstöra objekt, istället återanvänds de. För att återanvända objekt behöver ändringar ske. Exempelvis om ett zombieobjekt dör och ska återanvändas, måste dess position uppdateras till en ny plats. Även hälsan behöver uppdateras så att zombien inte är död längre. Det kan variera hur många ändringar som behövs för att återanvända objekt. Detta mönstret är speciellt bra när många objekt skapas och förstörs ofta (Unity Technologies 2022).

2.5 Relaterade arbeten

Liknande studier har gjorts som testar hur OOD jämfört med DOD hanterar många objekt. En av dessa är Turpeinen (2020) som genomförde ett experiment där OOD och DOD jämfördes mot varandra genom rendering av folkmassor i Unity. Två projekt skapades, en för varje paradigm. I projekten var allt identiskt förutom vald paradigm. Testerna bestod av att karaktärer med animationer skapades inom en area. Testerna utfördes på mobiltelefoner, där antalet karaktärer ökade med jämna intervaller tills programmets prestanda var under 30 frames per sekund (fps). Resultaten presenteras därefter i grafer som analyserades, vilket i slutändan resulterade i slutsatser. Turpeinen kom fram till att DOD presterade bättre än OOD, speciellt när scenerna innehöll många objekt med färre komponenter.

För Wingqvist, Wickstrom & Memeti (2022) skapades också två projekt där samma spel utvecklades med OOD och DOD. Testerna utfördes på olika datorer med olika operativsystem. Cirklar var placerade i en miljö och spelarens mål var att äta upp så många av dem som möjligt. Vissa av cirklarna stod helt stilla, vissa rörde på sig och andra var fiender som spelaren skulle undvika. Prestandan av programmet övervakades samtidigt som cirklarna ökade med jämna intervaller tills ett maxantal cirklar nåddes. Både OOD och DOD testades med och utan parallellisering. Resultatet presenterades i linjegrafer och stapeldiagram, vilket visade att DOD presterade bättre än OOD både med och utan parallellisering. DOD presterade bättre vid högre antal objekt, såg färre cache misses och hade ett maximum av 13,25 gånger bättre prestanda än OOD.

Turpeinen (2020) och Wingqvist, Wickstrom & Memeti (2022) var fokuserade på vilket paradigm som klarade av att hantera flest objekt. Ingen av studierna tar i åtanke hur OOD eller DOD presterar när objekt skapas och förstörs.

3 Problemformulering

Arbetet utgår från hur dataorienterad design (DOD) jämfört med objektorienterad design (OOD) presterar då många objekt ska hanteras, skapas och förstöras. Artefakten är uppbyggd i Unity där båda programmeringsparadigm kan ställas mot varandra med hjälp av komponentsystem. Programmeringsparadigmen testades genom att variera antalet objekt i scenen samt antalet objekt som skapades och förstördes. Under testerna övervakades prestandan som sedan användes för att skapa grafer.

3.1 Frågeställning

Frågeställningen som arbetet ämnar att undersöka är hur OOD jämfört med DOD presterar när många objekt måste hanteras, skapas och förstöras?

Prestanda är viktigt för spel, Janzen & Teather (2014) anser att spel borde minst uppnå 45 frames per sekund (fps) för att inte påverka spelaren. Frames per sekund kan också översättas till millisekunder, där 240 fps är ~4.2 ms och 180 fps är ~5.6 ms. Millisekunder visar hur lång tid det tog att exekvera en hel frame. Detta visar hur en liten ökning i millisekunder har stor påverkan på fps:en. I spel skapas och förstörs objekt, i vissa fall många gånger per sekund. Detta kan påverka prestandan negativt då objekt måste allokeras eller avallokeras ofta (Microsoft 2023, Unity Technologies 2025b). Därför är det viktigt att studera påverkan av allokeringar och hur dessa paradigm hanterar det.

Det har visats att DOD utnyttjar processorns cache bättre än vad OOD gör och kan därför hantera större mängder objekt (Wingqvist, Wickstrom & Memeti 2022). Syftet med arbetet är att ge en bättre uppfattning om det är fördelaktigt att använda DOD i spel med många objekt, samt när objekt skapas och förstörs ofta.

Hypotesen för arbetet är att OODOP kommer att prestera bäst eftersom objekt återanvänds istället för att allokeras på nytt. DOD kommer att prestera bättre än OOD för att cache användningen är bättre och DOD använder inte en garbage collector.

3.2 Metodbeskrivning

Metoden som valdes för att besvara frågeställningen var experiment. För att relatera arbetet till spelindustrin kommer Unity användas. I Unity kan både OOD och DOD användas vilket gör att dessa två programmeringsparadigm kan ställas mot varandra i samma miljö.

Testerna som genomfördes var på OOD, OODOP samt DOD där burst compile och parallellisering användes. Syftet med testerna var att efterlikna spel där objekt har en kort livslängd. Ett exempel på detta är *bullet hell*-spel där projektiler ofta skapas och förstörs. I testerna representerades projektilerna av cirklar. Antalet cirklar som användes i testerna var 1500, 5000, 10000 och 50000, vilket representerar det maximala antalet aktiva objekt i scenen. För omsättningen av cirklar (antal cirklar som skapades och förstördes varje frame) gick det från 150 till 1500 med ett intervall på 150 cirklar, vilket ger 10 samplingpunkter. Testerna gick igenom alla maxantal av aktiva cirklar med alla omsättningar av cirklar. Varje omsättning testades under 60 sekunder med en maxtid på 120 sekunder. Endast OODOP återanvände cirklar istället för att förstöra och skapa nya.

Datainsamlingen skedde genom övervakning av programmets prestanda genom att mäta hur lång tid varje frame tog att exekvera. Detta tog plats i update-loopen för att undvika

användandet av ett tredjepartsprogram då de måste valideras innan användning. Update-loopen går också igenom hela programmet och ger en tydlig bild av hur programmet presterar. Resultatet loggades till en textfil som sedan användes för att skapa grafer av.

Specifikationer för systemet som testen utfördes på:

Operativsystem: Windows 10

CPU: Intel i5-10400

CPU-L1: 384KB

CPU-L2: 1,5 MB

CPU-L3: 12MB

RAM: DDR4 32GB 2400MHz

GPU: ASUS NVIDIA GeForce RTX 3060

Turpeinen (2020) såväl Wingqvist, Wickstrom & Memeti (2022) inspirerade metoden för arbetet. Två projekt skapades, ett för varje programmeringsparadigm. Projekten hölls så identiska som möjligt, där den enda skillnaden var valda programmeringsparadigm. För arbetet mäts också hela programmets prestanda men till skillnad från Turpeinen kördes inte testerna tills prestandan var under en gräns. Istället kördes testerna tills en maxgräns av objekt var nådd. Liknande till Wingqvist, Wickstrom & Memeti (2022) användes parallellisering för DOD eftersom att paradigmet har fördelen av att vara enkel att parallellisera. OOD testades istället med och utan OODOP, vilket ofta används i samband med OOD när många objekt ska skapas och förstöras. Wingqvist, Wickstrom & Memeti (2022) presenterade sina resultat med hjälp av en linjegraf och liknande görs i detta arbete.

Eftersom att ingen av studierna tog upp hur OOD eller DOD presterar när objekt skapas och förstörs, ämnar detta arbetet att kontrastera de tidigare studierna genom att ta hänsyn till skapandet och förstörandet av objekt, men även jämförandet mellan OOD och DOD.

3.3 Metoddiskussion

Den valda metoden för arbetet är experiment då både Turpeinen (2020) och Wingqvist, Wickstrom & Memeti (2022) använder sig av samma metod. Eftersom ett flertal tester genomfördes där prestandan övervakades, bör en isolerad miljö användas, vilket experiment möjliggör.

Projekten försöker att efterlikna spel där många objekt skapas och förstörs ofta. Exempelvis *bullet hell-* eller *beat 'em up*-spel där objekt som fiender och projektiler inte finns speciellt länge. Det görs genom att ha många objekt i scenen samtidigt där ett antal av objekten omsätts. Genom att analysera omsättningen av objekt samt endast förstöra en andel av objekten i scenen kan experimentet mer efterlikna spel då inte alla objekt som skapas kommer finnas för evigt. För att hålla projekten inom tidsramen implementerades ingen interaktion mellan objekten och miljön. Användningen av enkel grafik skedde också av samma anledning, och för att fokusera mer på experimentet.

Som Fedoseev, Askarbekuly, Uzbekova & Mazzara (2020) nämner i sin studie bör medelvärdet användas på grund av att olika system kan sporadiskt störa resultaten. Eftersom hela programmets prestanda mäts kan andra delar av Unity påverka resultatet. För att minimera påverkan av dessa system beräknades ett medelvärde. Nackdelen med medelvärdet

är att spridningen inte visas och därför beräknades även standardavvikelsen. Även om experimenten utvecklas i Unity kan andra program dra nytta av resultaten, eftersom paradigmen inte är begränsade till enbart Unity eller spel.

Fedoseev m.fl. (2020) påpekar att DOD har fördelen av att vara enklare att parallellisera samt att jobbet sprids ut enhetligt över trådarna. Wingqvist, Wickstrom & Memeti (2022) nämner även hur olika trådar kan samtidigt utnyttja en delad cache, något som DOD skulle vara bättre på då data effektivt placeras i minnet kontinuerligt. För DOD användes parallellisering eftersom den har fördelen av att vara enklare att parallellisera. OOD använde istället OODOP, en fördel OOD har och som ofta implementeras för att undvika prestandapåverkan av allokeringar.

Något som blir viktigt är att projekten och scenerna är så identiska som möjligt, skriver Fedoseev m.fl. (2020). Detta är för att minska påverkan av skillnader i resultatet och för att säkerställa att rätt saker mäts. Genom att ha identiska scener och samma miljö försöker arbetet att isoleras den enda skillnaden, vilket är användandet av olika programmeringsparadigm.

På grund av att mätningar hände i update-loopen blir det olika antal mätvärden som används för medelvärdet. Det leder till att ju sämre prestandan är, desto färre mätvärden kommer att användas för medelvärdet, vilket i sin tur gör att resultaten blir mindre pålitliga.

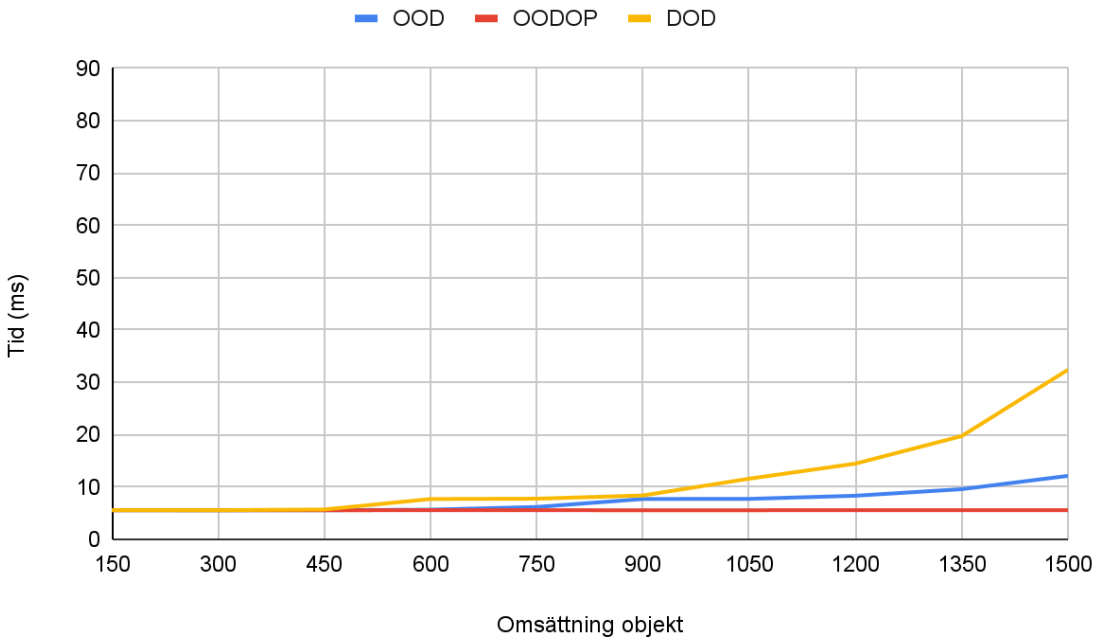
4 Undersökning

I detta kapitlet kommer artefakten och resultatet presenteras samt kommer resultatet att analyseras.

4.1 Artefakt

Två projekt skapades, det vill säga ett projekt för varje programmeringsparadigm. Projekten skapades i Unity för att relatera arbetet mer mot spel samt mot en större grupp inom spelindustrin. Unity har också stöd för både OOD och DOD inom samma miljö vilket hjälpte med skapandet och mätandet. Versionen av Unity som användes var 6000.0.27f1 och för DOTS användes version 1.3.9. I projekten skapades cirklar slumpmässigt inom en area. Varje frame omsätts ett antal cirklar. Detta sker genom att först förstöra slumpmässigt utvalda cirklar och därefter skapades lika många nya. Omsättningen av objekt hände med jämna intervaller, varje 20 ms. Projekten var i 2D och det fanns ingen interaktion mellan objekten eller miljön genom exempelvis fysik. I OOD-projektet var även OODOP implementerat och i detta fallet förstördes eller skapades inte cirklar, istället avaktiverades och aktiverades dem.

4.2 Resultat



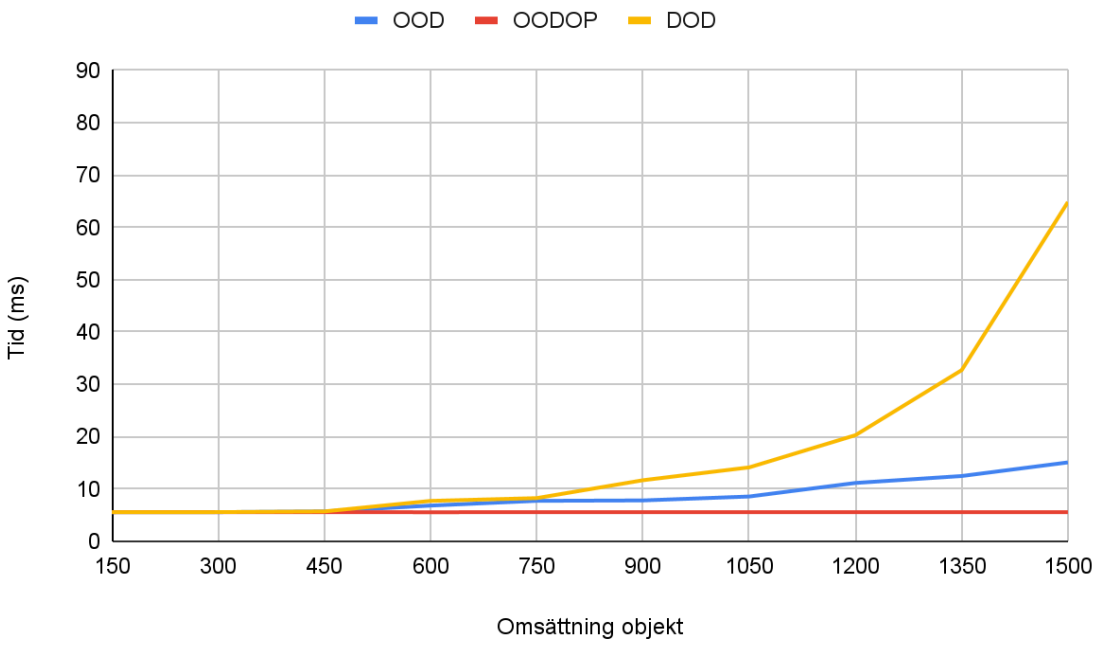
Figur 4, visar medelvärdet för 1500 objekt i scenen för OOD, OODOP och DOD vid olika omsättningar av objekt.

För varje test omsätts objekt 3000 gånger, vilket tillät garbage collector:n att rensa oanvänd data flera gånger. Som figur 4 visar presterade OODOP bäst och låg lite över 5 ms under alla omsättningar av objekt. OODOP såg inte heller några stora prestanda förändringar. Till en början ligger alla implementationer nära varandra tills 450 objekt

i omsättning då DOD börjar prestera värre medans det tar tills 750 objekt i omsättning innan OOD börjar prestera värre.

I figur 4 såg OOD som värst 1,18 gånger värre prestanda än OODOP medans DOD presterade 4,8 gånger värre än OODOP. Båda dessa var vid 1500 objekt i omsättning.

Prestanda med 5000 objekt i scenen

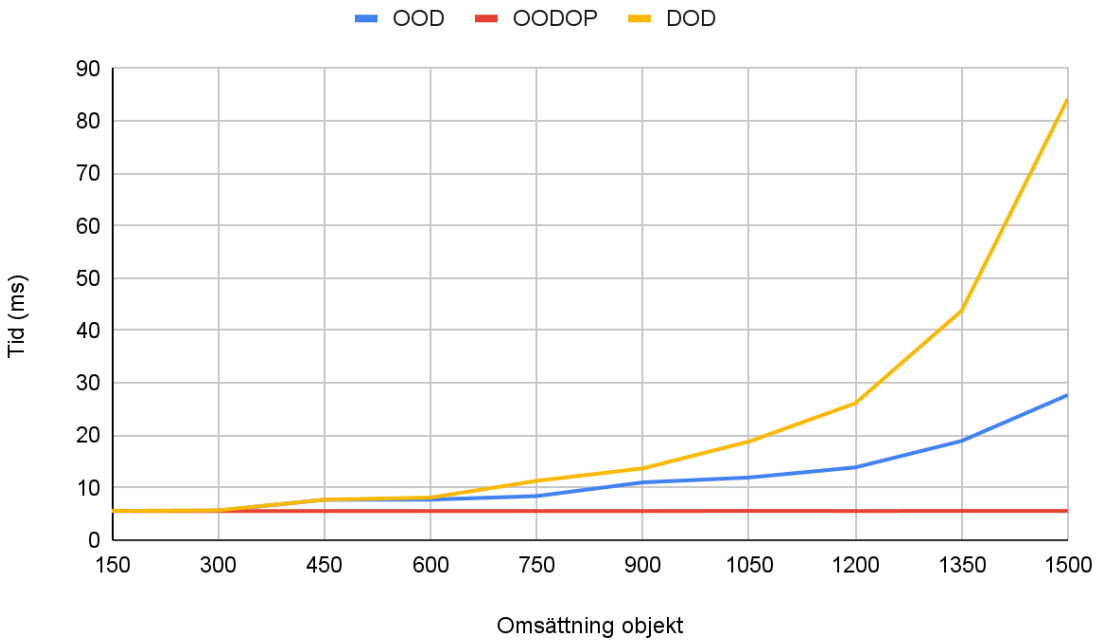


Figur 5, visar medelvärdet för 5000 objekt i scenen för OOD, OODOP och DOD vid olika omsättningar av objekt.

I figur 5 ses ett liknande resultat till figur 4. OODOP ligger kvar vid ~5 ms och visar ingen märkbar prestanda förändring. Både OOD och DOD börjar att prestera värre vid 450 objekt i omsättning och presterar liknande tills 750 objekt i omsättning innan DOD börjar att prestera sämre.

Som figur 5 visar presterade både OOD och DOD värst vid 1500 objekt i omsättning. OOD hade 1,71 gånger värre prestanda och DOD hade 10,6 gånger värre prestanda än OODOP.

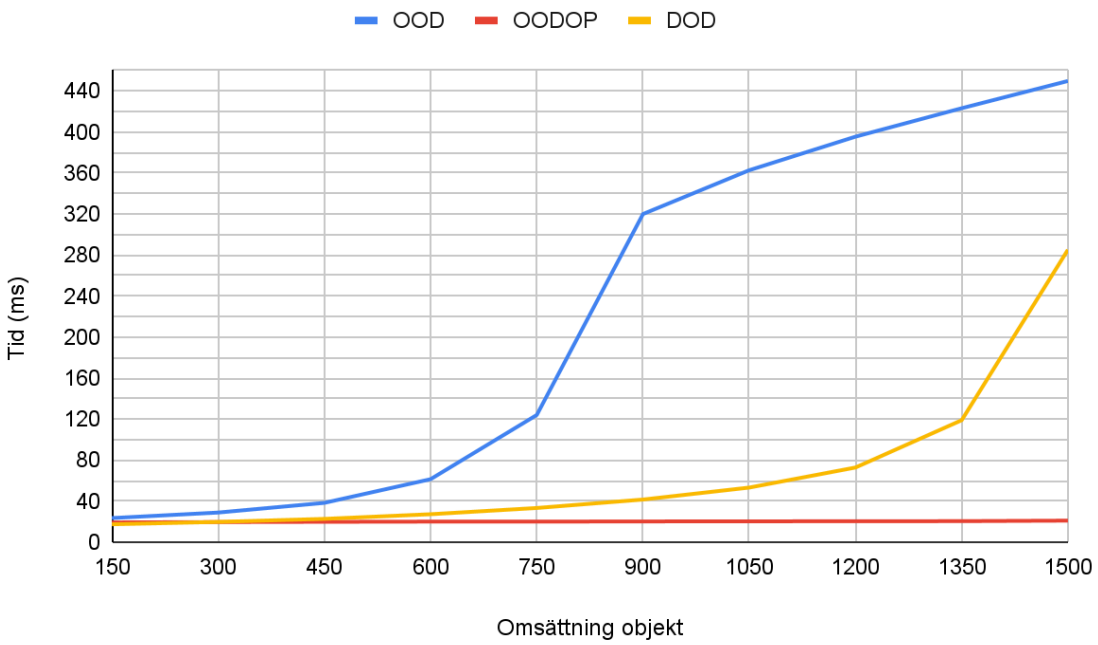
Prestanda med 10000 objekt i scenen



Figur 6, visar medelvärdet för 10000 objekt i scenen för OOD, OODOP och DOD vid olika omsättningar av objekt.

Även figur 6 ser liknande resultat till figur 4 och 5. OODOP presterar fortfarande bäst med ~5 ms och visar ingen märkbar prestanda förändringar över olika omsättningar objekt. Både OOD och DOD försämras vid 300 objekt i omsättning och håller liknande resultat fram tills 900 då DOD börjar att prestera sämre.

Liknande vid figur 6 presterar både OOD och DOD värst vid 1500 objekt i omsättning. Som värst ser OOD 3,9 gånger värre prestanda medans DOD ser 14,1 gånger värre prestanda än OODOP.



Figur 7, visar medelvärdet för 50000 objekt i scenen för OOD, OODOP och DOD vid olika omsättningar av objekt.

För figur 7 har y-axeln fått ett 5 gånger större maxvärde. Vid 50000 objekt i scenen har OODOP ökat från ~5 ms till ~20 ms. OODOP ser ingen skillnad mellan olika omsättningar av objekt. Här börjar DOD prestera bättre än OOD över alla omsättningar objekt. OODOP och DOD ser en liknande början ända upp till 450 objekt i omsättning innan DOD börjar att prestera sämre.

Även här presterar OOD och DOD värst vid 1500 objekt i omsättning. OOD presterade 20,7 gånger värre jämfört med OODOP. DOD såg som värst 12,6 gånger värre prestanda än OODOP, vilket är en mindre ökning än i figur 6.

4.3 Analys

Hypotesen för detta arbetet var att OODOP skulle prestera bäst då allokeringar endast skulle hända i början samt att objekten endast renderade en cirkel. DOD skulle prestera bättre än OOD eftersom DOD är bättre på att hantera större mängder objekt samt att ingen garbage collector användes. OOD skulle prestera sämst på grund av mängden objekt i scenen samt att en garbage collector användes. Detta visade sig vara delvis rätt.

Resultatet visade att OODOP presterade bäst och hade en stabil linje som visade hur väl den kunde hantera att omsätta objekt då all allokering hände i början. För 1500 till 10000 objekt i scenen såg OODOP ingen skillnad i prestanda då andra Unity system överskuggade påverkan av OODOP. Detta skedde på grund av att hela programmets prestanda övervakades och inte bara implementationerna. Det var först vid 50000 objekt som OODOP presterade värre, runt 20 ms från ~5 ms. Även OOD hamnade runt 23ms som lägst vid 50000 objekt. Ingen av OOD eller OODOP går under 20 ms och objektens enda belastning på programmet var rendering. Det kan då konstateras att renderingen är anledningen till ökningen för OODOP mellan

10000 till 50000 objekt i scenen. Även i Unitys prestandaprofilerare kan påverkan av renderingen ses.

För DOD stämde inte hypotesen. DOD presterade sämst vid färre antal objekt (under 50000) då OOD och OODOP kunde hantera allokeringarna bättre. Mellan OOD och DOD sågs en stor skillnad där DOD som minst presterade 4,8 gånger värre och som mest 14,1 gånger jämfört med OODOP vid 1500 objekt i omsättning. För OOD sågs minst 1,18 gånger värre och som mest 20,7 gånger värre prestanda jämfört med OODOP vid 1500 objekt i omsättning. Det är enbart vid lägre objekt i omsättning som OOD och DOD ligger nära varandra, men eftersom graferna inte går under ~5 ms (på grund av andra Unity system) är det svårt att se ifall detta faktiskt stämmer.

För figur 4-6 presterar OOD bättre än DOD. Detta tyder på att skapandet och förstörandet av objekt påverkar DOD mer än OOD som då effektivare kan hantera omsättningen. Men i figur 7 visar DOD sin styrka genom att vara bättre på att hantera större mängder objekt, även om den är mindre effektiv på att hantera skapandet och förstörandet. Med 50000 objekt i scenen och vid alla omsättningar presterade DOD bättre än OOD. Detta visar att antalet objekt i scenen påverkar OOD mer än DOD. I graferna kan också DOD ses förbättras ju fler objekt som läggs till i scenen. Det vill säga att DOD presterar liknande till OOD längre ju fler objekt som finns i scenen innan de drar iväg från varandra.

Vid 50000 objekt i scenen kan även OOD ses gå från ~120 ms vid 750 objekt i omsättning till ~320 ms vid 900 objekt i omsättning. Eftersom mätningen hände i update-loopen blev det mindre antal mätvärden desto värre prestandan av programmet blev vilket orsakade den kraftiga ökningen. Detta är en svaghet med metoden som tas upp i mer detalj i diskussionen. Objekten kan som mest uppdateras varje 300 ms, vilket innebär att om en frame tar 600 ms, kommer objekten att uppdateras två gånger på en frame. Som visas i figur 7 efter 300 ms ökar OOD inte lika kraftigt, vilket beror på den här maxtiden som skapar fler mätvärden.

Genom att öka allokeringen av objekt, kan en påverkan ses i resultatet där färre antal objekt som omsätts presterar bättre än när fler objekt omsätts. Det vill säga att ju fler allokeringar och avallokeringar som sker, desto sämre blir prestandan. Det är också därför OODOP presterade bättre, eftersom all allokering hände i början samt att inga objekt avallokerades.

Standardavvikelsen beräknades och visade att spridningen ökade med antalet omsättningar. Vilket är att förväntas eftersom när prestandan blir sämre, kommer spridningen att öka. Men det fanns problem med spridningens pålitlighet, vilket kommer att diskuteras mer ingående i diskussionen.

Eftersom objekten inte interagerar med miljön, hade aktiveringen och avaktiveringen av objekt i OODOP ingen märkbar påverkan på resultatet, vilket tyder på brister med validiteten. När objekt återanvänds genom OODOP i spel, behöver de på något sätt ändras eller återställas, vilket testerna inte representerade väl. För att åtgärda detta utfördes ytterligare ett test, med målet att bättre representera spel som använder OODOP. I testet hade objekt fysikkomponenter, vilket gav fler parametrar att återställa när objektet återanvändes. Testet utfördes endast för 10000 objekt i scenen, och 1500 objekt i omsättning. OODOP med fysik såg 1,25 gånger sämre prestanda jämfört med OODOP utan fysik. Detta visar att återanvändningen fortfarande har en påverkan på prestandan. Trots detta presterar OODOP med fysik fortfarande bättre än OOD och DOD.

5 Sammanfattning och diskussion

5.1 Sammanfattning

I arbetet har objektorienterad design (OOD) jämförts med dataorienterad design (DOD) när många objekt måste hanteras, skapas och förstöras. Det är viktigt för ett realtidsprogram som spel att hålla programmets prestanda runt 45 frames per sekund eller högre, annars kan det påverka spelaren (Janzen & Teather 2014). Prestandan kan även påverkas när objekt skapas och förstörs då allokeringar kan ha stor påverkan på prestandan (Microsoft 2023).

Två projekt har tagits fram för arbetet, ett för OOD och ett för DOD. Projekten skapades med tanken att de skulle vara så identiska som möjligt, där den enda skillnaden är vald programmeringsparadigm. I testerna skapades cirklar slumpmässigt inom en area. Varje frame omsätts objekt genom att först ta bort ett antal av dem och sedan skapa lika många nya. Genom att mäta tiden varje frame tog att exekvera under en viss tid, kunde medelvärdet samt standardavvikelsen beräknas som visade programmets prestanda. Testerna som utfördes var för OOD, DOD och OOD med object pooling (OODOP). Mätningen skedde i update-loopen vilket gjorde att tester som hade sämre prestanda fick mindre antal mätvärden. Detta leder till att sämre prestanda gör resultatet mindre pålitligt.

Resultaten visade att OODOP presterade bäst eftersom all allokering hände i början samt att ingen avallokering förekom. För OODOP syntes ingen skillnad när omsättningen ökade, då objekt kunde direkt återanvändas utan ändringar eller att återställas. Att objekten kan direkt återanvändas utan att behöva några ändringar är osannolikt i spel. För att mer likna användningen av OODOP inom spel skapades ytterligare ett test. Detta testet visade att även vid fler ändringar presterade OODOP fortfarande bättre än OOD och DOD. Men testet visade också att fler ändringar resulterade i sämre prestanda. Vid färre antal objekt presterade OOD bättre än DOD. Detta är på grund av hur allokeringar påverkade DOD mer, vilket ledde till sämre prestanda. Med större mängder objekt började OOD prestera sämre. Resultatet visade att DOD kunde bättre hantera större mängder objekt än OOD. För allokeringar var det tvärtom, OOD påverkades mindre prestandamässigt av allokeringar än DOD. I resultatet kan även omsättningen ses ha en påverkan, ju fler objekt som omsätts desto sämre blev prestandan.

5.2 Diskussion

Att välja mellan OOD och DOD kan vara svårt eftersom båda programmeringsparadigm har sina för- och nackdelar. Dessa paradigm behöver dock inte användas för hela projektet och kan istället användas per system. Det betyder att båda paradigmen kan användas inom samma projekt. Som Llopis (2009) konstaterar, kan OOD fortfarande vara relevant att använda i exempelvis GUI-system. Som Baylis (2022) nämner, används OOD mer än DOD. Det är på grund av att DOD inte lärs ut i många utbildningar samt att det finns övergripande brister inom studier av DOD. Llopis (2009) påpekar samma nackdel med DOD och hur programmerare som kommer från en bakgrund med OOD har det svårare att plocka upp DOD. Men som Llopis (2009) och Nystrom (2021) skriver är fördelarna av DOD att programmet får bättre prestanda och blir mer modulärt.

Resultatet visade att OOD presterade bättre än DOD vid färre antal objekt. För Turpeinen (2020) såg DOD en bättre prestanda än OOD i nästan alla tester. Även Wingqvist, Wickstrom & Memeti (2022) märkte en prestandaförbättring redan vid 5000 objekt. Det betyder att

allokeringar hade stor påverkan på prestandan eftersom DOD endast presterade bättre vid 50000 objekt. Det är genom strukturella förändringar, som måste hanteras på huvudtråden, och att all parallellisering behöver invänta de förändringarna, som gör att DOD och då DOTS presterar värre (Unity Technologies 2025a). Det betyder också att OOD och garbage collector:n är mer effektiv på att hantera allokeringar.

Enligt resultatet bör OODOP användas när många objekt skapas och förstörs. OODOP har dessutom fördelen att vara enklare att implementera eftersom OOD är mer använt än DOD. Detta gör att fler personer kan utnyttja prestandaförbättringarna som OODOP medför, utan att behöva lära sig ett helt nytt paradigm, vilket kan vara svårt och tidskrävande. Wingqvist, Wickstrom & Memeti (2022) tar även upp hur spel ofta inte har över 5000 objekt i en scen. Det är även osannolikt att spel allokerar och avallokerar 1500 objekt varje frame. Även för 5000 objekt bör OODOP eller bara OOD användas som presterar bättre än DOD.

För allmänt användning av DOD bör det användas för att hantera många objekt där få allokeringar förekommer. Turpeinen (2020) och Wingqvist, Wickstrom & Memeti (2022) visar att DOD är bättre på att hantera större mängder objekt än OOD, men nämner inget om allokeringar. Detta arbetet har visat att DOD inte alltid presterar bäst. Det är när många objekt behöver allokeras som DOD faller, OOD och OODOP kan tydligt hantera dessa allokeringar bättre.

Garbage collector:n för Unity är optimerad då den hantera avallokeringar över flera frames istället för alla på en, vilket har varit en fördel för OOD. Turpeinen (2020) kommer fram till att vid färre komponenter på objekten presterar DOD bättre än OOD. I testerna hade objekten inte många komponenter vilket har varit en fördel för DOD. Att mätningen tog plats i update-loopen har haft en negativ påverkan på resultatet. Sämre prestanda resulterade i att färre mätvärden användes för medelvärdet. Resultatets trovärdighet har påverkat på grund av detta, där värre prestanda ger mindre trovärdiga resultat. Men även vid de resultat som presterade sämst användes minst 100 mätvärden för medelvärdet. Det nämndes att standardavvikelsen beräknades, men att det fanns problem med pålitligheten. Eftersom hela programmets prestanda mättes, påverkades spridningen av andra Unity-system. Även här påverkade problemet med update-loopen, vilket gjorde att spridningen ökade kraftigt vid färre mätvärden. Detta gjorde att spridningen stämde dåligt överens med medelvärdet, speciellt för DOD.

Beroende på projektets komplexitet kan resultaten variera. Precis som i testerna för OODOP, med fler ändringar blev prestandan sämre. Det skulle kunna innebära att prestandan påverkas mer ju fler ändringar som behövs, men detta behöver däremot mer forskning. Allokeringar kan också prestera sämre ju mer data ett objekt innehåller. Detta kan leda till sämre prestanda då större datamängder behövs hanteras. Dessutom kan processorns cache bara lagra en viss mängd data. Om komponenter innehåller mycket data får färre av dem plats i cachen, vilket leder till fler cache misses och därmed sämre prestanda.

5.3 Samhälleliga och etiska aspekter

Företag kommer att ha användning av information för att optimera sina realtidsprogram. Detta kan öka tillgängligheten av programmet då optimeringar kan tillåta äldre system att köra programmet.

Optimeringar kommer också att öka prestandan av programmet, vilket i sin tur påverkar

hårdvaran. Som Gorman (2022) påpekar förbättrar DOD prestandan och processorn kan då snabbare jobba igenom arbetet. Det gör att processorn snabbare kan gå in i viloläge och då förbruka mindre el. Detta är inte något som bara DOD gör, utan all optimering snabbar upp arbetet för processorn och resulterar i att mindre el behövs för samma arbete.

5.4 Framtida arbete

OODOP visade ingen skillnad mellan olika mängder omsättningar. När ytterligare ett test utfördes med fysik komponenter visade det sig att OODOP med fysik ökade 1,25 gånger jämfört med OODOP utan fysik. Det hade då varit intressant att inte bara testa OODOP utan också OOD och DOD kopplat till ett större spelsammanhang. Det skulle då finnas fler system som påverkade resultaten och visade ett mer kopplat sammanhang till spel.

Under testerna skapades och förstördes objekt inom samma frame. Det hade varit intressant att separera på dem för att se vilket av dem som påverkade prestandan mest. Resultaten hade då kunnat visa vilken av allokering eller avallokering som påverkade paradigmet mest.

Referenser

Aniche, M. (2024). *Simple Object-Oriented Design.* Manning Publications.

Bayliss, J.D. (2022). *Developing Games with Data-Oriented Design*. doi: 10.1145/3524494.3527626

Fedoseev, K., Askarbekuly, N., Uzbekova, E. & Mazzara, M. (2020). *A Case Study on Object-Oriented and Data-Oriented Design Paradigms in Game Development*. doi:10.13140/RG.2.2.16657.66405.

Gorman, J. (2022). *Notes: Data-Oriented Design.* Tillgänglig på internet: https://jasongorman.uk/notes/data-oriented-design/ [Hämtad Feburari 7, 2025]

Janzen, B.F. & Teather, R.J. (2014). *Is 60 FPS better than 30? The impact of frame rate and latency on moving target selection*, I CHI EA '14: CHI '14 Extended Abstracts on Human Factors in Computing Systems. United States, New York 26 april - 1 maj 2014. s. 1477–1482. doi:10.1145/2559206.2581214.

Llopis, N. (2009). *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)*. Tillgänglig på internet: https://gamesfromwithin.com/data-oriented-design [Hämtad Februari 7, 2025]

Lowy, J. (2005). *Programming .NET Components, 2nd Edition*. 2 uppl., O'Reilly Media, Inc.

Microsoft. (2023). *Fundamentals of garbage collection*. Tillgänglig på internet: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals [Hämtad Februari 25, 2025]

Nystrom, R. (2021). *Game Programming Patterns*. Tillgänglig på internet: *https://gameprogrammingpatterns.com/data-locality.html* [Hämtad Februari7, 2025]

Sharvit, Y. (2022). *Principles of Data-Oriented Programming*. Tillgänglig på internet: https://blog.klipse.tech/dop/2022/06/22/principles-of-dop.html [Hämtad Mars 12, 2025]

Turpeinen, M. (2020). *A Performance Comparison for 3D Crowd Rendering using an Object-Oriented system and Unity DOTS with GPU Instancing on Mobile Devices*, Masteruppsats, KTH. Tillgänglig på internet: https://kth.diva-portal.org/smash/get/diva2:1467022/FULLTEXT01.pdf [Hämtad Januari 21, 2025]

Wingqvist, D., Wickstrom, F. & Memeti, S. (2022). *Evaluating the performance of object-oriented and data-oriented design with multi-threading in game development*. I 2022 IEEE Games, Entertainment, Media Conference (GEM). Barbados, Saint Michael 27-30 november 2022. doi:10.1109/GEM56474.2022.10

Unity Technologies. (2025a). *Entity Component System concepts*. Tillgänglig på internet: https://docs.unity3d.com/Packages/com.unity.entities@1.3/manual/concepts-intro.html [Hämtad Februari 4, 2025]

Unity Technologies. (2024). *Introduction to the Data-Oriented Technology Stack for advanced Unity developers*. Tillgänglig på internet: https://unity.com/resources/introduction-to-dots-ebook [Hämtad Februari 4, 2025]

Unity Technologies (2017). *Introduction to GameObjects.* Tillgänglig på internet: https://docs.unity3d.com/6000.0/Documentation/Manual/GameObjects.html [Hämtad April 1, 2025]

Unity Technologies. (2022). *Introduction to Object Pooling*. Tillgänglig på internet: https://learn.unity.com/tutorial/introduction-to-object-pooling# [Hämtad Februari 3, 2025]

Unity Technologies. (2025b). *Memory in Unity introduction*. Tillgänglig på internet: https://docs.unity3d.com/6000.0/Documentation/Manual/performance-memory-overview.html [Hämtad Februari 7, 2025]

Unity Technologies. (2025c). *Unity*. Tillgänglig på internet: https://unity.com/ [Hämtad Januari 20, 2025]