Examensarbete

HÖGSKOLAN I SKÖVDE

ATT UPPTÄCKA DOLDA OBJEKT I SPEL

En undersökning av *occlusion culling* och dess påverkan i spelscener

TO DETECT HIDDEN OBJECTS IN GAMES

An analysis of *occlusion culling* and its affect in game scenes

Examensarbete inom Datalogi
Grundnivå/ 30hp
Vårtermin 2017

Jakob Eriksson

Examinator: Henrik Engström

Sammanfattning

Att detektera och eliminera dolda ytor är viktigt när grafiska applikationer blir mer komplexa och detaljerade. En teknik som används till det är *occlusion culling* och olika typer av algoritmer har tagits fram för att möjliggöra den tekniken. *Occlusion culling* innebär att hitta objekt som har stor potential att dölja delar av en scen. Dessa objekt kallas för *occluders*. I den här rapporten så undersöks möjligheten att använda olika stor mängd av *occluders* i en algoritm och hur det påverkar olika typer av spelscener. Tre olika spelscener har satts upp i Unity som skiljer sig i hur de är uppbyggda. Effekten av *occlusion culling* i scenerna visar på att det kan vara en teknik bättre lämpad för vissa typer av spel. Rapporten visar också på en möjlighet att forska vidare på valet av en lämplig mängd med *occluders*.

Nyckelord: *Occlusion culling, occluders*, spelscen, Unity

Innehållsförteckning

1 Introduktion

För att kunna skapa komplexa scener där det finns många objekt och där scenen kan flyta på i realtid har det forskats om tekniker för att eliminera objekt som inte är synliga. En av de teknikerna är *occlusion culling* och det är den tekniken som den här rapporten fokuserar på. När en grafisk applikation renderar en scen är det möjligt att objekt är dolda av varandra. En *occlusion culling*-algoritm går ut på att detektera de objekt som är dolda så att de dolda objekten inte behöver renderas (Chrysanthou, Cohen-Or, Durand & Silva, 2003; Coorg & Teller, 1996). Vanligtvis så används en delmängd av objekten i en scen som *occluders*. Denna delmängd bör bestå av de objekt som har störst potential att dölja andra objekt och enligt Bittner, Havran och Slavík (1998) kan det vara fördelsaktigt att bestämma potentiella *occluders* i ett förbehandlingssteg. Då är det viktigt att undersöka om en förändring i delmängden med *occluders* påverkar hur många objekt som är dolda i en scen. Olika scener kan ha stora variationer i hur de är uppbyggda så det är intressant att testa flera olika scener där förutsättningarna för *occlusion culling* inte är likadana. Forskning som finns för *occlusion culling* har fokuserat mest på stadsscener och inomhusmiljöer där det ofta finns lämpliga objekt som kan användas som *occluders* (Cohen, Hoff, Hudson, Lin, Manocha & Zhang, 1997; Manocha, Salomon & Yoon, 2003).

Det här projektet undersöker en *occlusion culling*-algoritm i tre typiska spelscener för att kunna förstå påverkan tekniken kan ha på olika typer av spel. För varje scen så implementerades en automatisk kameraåkning som simulerar hur kameran vanligtvis förflyttar sig. För att få ett mätbart resultat kördes varje scen med tre olika mängder *occluders*. För varje körning sparades den genomsnittliga tiden av overheaden för *occlusion culling*-algoritmen och den genomsnittliga renderingstiden. Antalet renderade polygoner sparades ner från körningen som sedan sammanställdes i grafer så det går att se hur resultatet för de olika scenerna skiljer sig åt.

2 Bakgrund

Det som gör att bl.a. utvecklare vill undersöka grafikalgoritmer är att de hela tiden vill att system ska använda så lite datorkraft som möjligt för att kunna göra mer. De mål som utvecklare vill uppnå i realtidsrendering är att ha många bilder per sekund, en högupplöst display och mer realistiska objekt i miljön.

Bilder per sekund (*frames per second* eller förkortat *FPS* på engelska) är som namnet säger utritningsfrekvensen av miljön per sekund. Akenine-Möller och Haines (2002) underströk att det är viktigt att ha en stadig bilder per sekundfrekvens för att ha en mjuk och kontinuerlig erfarenhet. Idealet är att system renderar bilder med samma frekvens som sin monitors frekvens. Har program en låg bilder per sekundfrekvens så kan det kännas hackigt och med hjälp av algoritmer för att skära bort de objekt som är onödiga att rita ut kan det förbättra programmets bilder per sekundfrekvens. Att ha en hög bilder per sekundfrekvens behövs enligt Power och Rubino (2008) för att ge en känsla av sömlös, flytande interaktivitet.

En av anledningarna till att programs bilder per sekundfrekvens ökar är att objekt som inte renderas inte behöver gå igenom den grafiska pipelinen där flera operationer utförs så som transformationer, ljusberäkningar och texturmappning m.fl. (Pantazopoulos & Tzafestas, 2002).

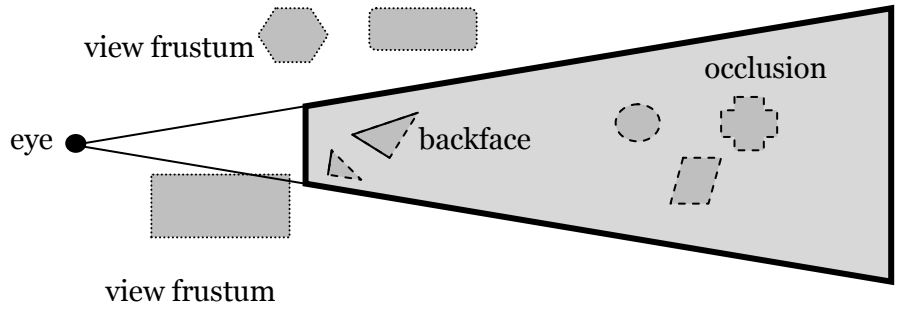
2.1 Culling-tekniker

Cull betyder "att ta bort från en flock" (Akenine-Möller & Haines, 2002) och det är precis vad utvecklare gör när de implementerar *culling*-tekniker, de tar bort det från scenen som inte är nödvändigt att rita ut.

En av de vanligare teknikerna som används är enligt Akenine-Möller och Haines (2002) t.ex. *backface culling* som eliminerar polygoner som är bortvända från kameran. De beskriver olika sätt att uppnå det och algoritmer kan t.ex. utföra operationer för att kontrollera att en ytas normal är bortvänd från kameran.

En annan teknik som används är *view frustum culling* och den tekniken är lite mer komplex då den tar bort alla objekt som inte ligger inom kamerans *view frustum*. Algoritmen kan dra nytta av att utvecklaren använder sig av rumsindelningsstrukturer för att få en effektiv algoritm (Assarsson och Möller, 2000). Olika rumsindelningsstrukturer tas upp i nästa kapitel.

De teknikerna leder fram till *occlusion culling*. Innebörden av den tekniken är att utvecklare inte vill rita ut de objekt som döljs av andra objekt för att det skulle vara onödigt. Power och Rubino (2008) uttrycker sig att program inte ska göra beräkningar på det som inte är synligt. Chrysanthou, Cohen-Or, Durand och Silva (2003) skriver att *occlusion culling* är global eftersom den berör ömsesidiga förhållande mellan polygoner och är därför mer komplex än *backface culling* och *view frustum culling*. Figur 1 nedan demonstrerar hur de tre olika teknikerna ser ut från fågelperspektiv när en kamera ser ut över en miljö.

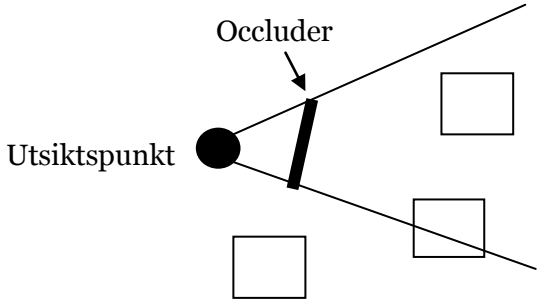


Figur 1 En bild över olika *culling*-tekniker (efter Akenine-Möller & Haines, 2002).

En annan teknik som används är *portal culling* som är användbar när scenen innehåller flera rum. Algoritmen delar in scenen i flera celler som ofta är rum eller korridorer och där dörrar eller fönster är portalerna mellan dem. Ett hus är ett bra lämpat objekt att dela upp med portaler och celler enligt Aila och Miettinen (2004). De skriver att spel ofta använder sig av den tekniken när scener består mestadels av inomhusmiljöer. Genom att använda sig av sitt *view frustum* kan algoritmen se vilka celler som är aktiva och systemet kan skippa att rendera de som inte är aktiva. Genom att associera objekt till de olika cellerna kan algoritmen också avgöra vilka objekt som behövs ritas ut genom att kontrollera de celler som är aktiva. Enligt Akenine-Möller och Haines (2002) är *portal culling* en variant av *occlusion culling*-tekniken men de skiljer på dem för att *portal culling* är en sådan viktig teknik.

2.2 *Occluders*

En *occluder* är ett objekt som används i en *occlusion culling*-algoritm för att dölja andra objekt. De som skrivit algoritmer har därför lagt tid på att hitta sätt att välja ut objekt som är lämpliga som *occluders*. Från en godtycklig synvinkel är det ett fåtal polygoner som döljer flest objekt och att använda alla polygoner skulle öka overheaden av algoritmen utan att finna fler dolda objekt (Coorg & Teller, 1996). Aila och Miettinen (2004) skriver att inte alla synliga objekt är meningsfulla *occluders*. En klocka på väggen är överflödig eftersom väggen är en effektiv *occluder* och glas på ett bord är för små och p.g.a. deras transparens kommer de inte dölja något även om kameran ser på dem på nära håll. Av den anledningen är det generellt en optimering att välja en delmängd av alla synliga objekt som *occluders*. De nämner också att *occluders* närmare kameran oftare ger ett bättre resultat än de som är långt ifrån kameran.



Figur 2 Exempel på hur en *occluder* kan se ut i en scen (efter Cohen et al., 1997).

Chrysanthou et al. (2003) undersöker och sammanställer flera metoder som undersöker *occlusion culling* och skriver att metoderna använder antingen alla objekt som *occluders* eller en delmängd av objekten i scenen som *occluders*, vanligtvis stora objekt. Bittner, Havran och Slavík (1998), Cohen, Hoff, Hudson, Lin, Manocha och Zhang (1997) samt Coorg och Teller (1996) använder sig av rymdvinkeln för att väja ut *occluders* som har bra möjlighet att dölja en stor del av scenen. På så sätt får de ut ett estimat på hur stor del en *occluder* täcker från en utsiktspunkt. I figur 3 beskrivs formeln för att få ett rimligt estimat på rymdvinkeln av en *occluder* från en godtycklig position i scenen. A är arean av polygonen, N är normalvinkeln för polygonen, V är vinkeln som kameran är riktad och D är avståndet från kameran till mitten på polygonen.

$$\frac{-A(\vec{N} \cdot \vec{V})}{\|\vec{D}\|^2}$$

Figur 3 Formel för ett estimat av rymdvinkeln för en polygon (efter Coorg och Teller, 1997).

De som har skrivit tekniker för att räkna ut konservativ synlighet klassificerar teknikerna som objektrums- och bildrumsalgoritmer (Cohen et al., 1997). Sättet som Chrysanthou et al. (2003) benämner det är bildprecisions- och objektprecisionsmetoder. Objektprecisionsmetoder använder obearbetad data för synlighetsförfrågningar och bildprecisionsmetoder använder den diskreta representationen av objekt som delats in i fragment under rasteriseringen. Pantazopoulos och Tzafestas (2002) uttrycker sig att ett kännetecken för objektrumsalgoritmer är att beräkningar görs i tredimensionella rummet. För bildrumsalgoritmer så görs beräkningar genom grundläggande bildbehandling. Det finns fördelar och nackdelar med algoritmerna och tidigare studier har noterat en del av dem. Vanligtvis så brukar objekt vara dolt av flera *occluders* kombinerade effekt. Att föra samman *occluders* så att synlighet på objekt testas mot sammanförda *occluders* är enligt Aila och Miettinen (2004) problematiskt för objektrumsalgoritmer men är enklare för bildrumsalgoritmer. Cohen et al. (1997) skriver också om problemen med att föra samman *occluders* i sin objektrumsalgoritm. Däremot så säger de att en fördel är att deras algoritm inte behöver en avancerad grafisk pipeline för att göra beräkningar.

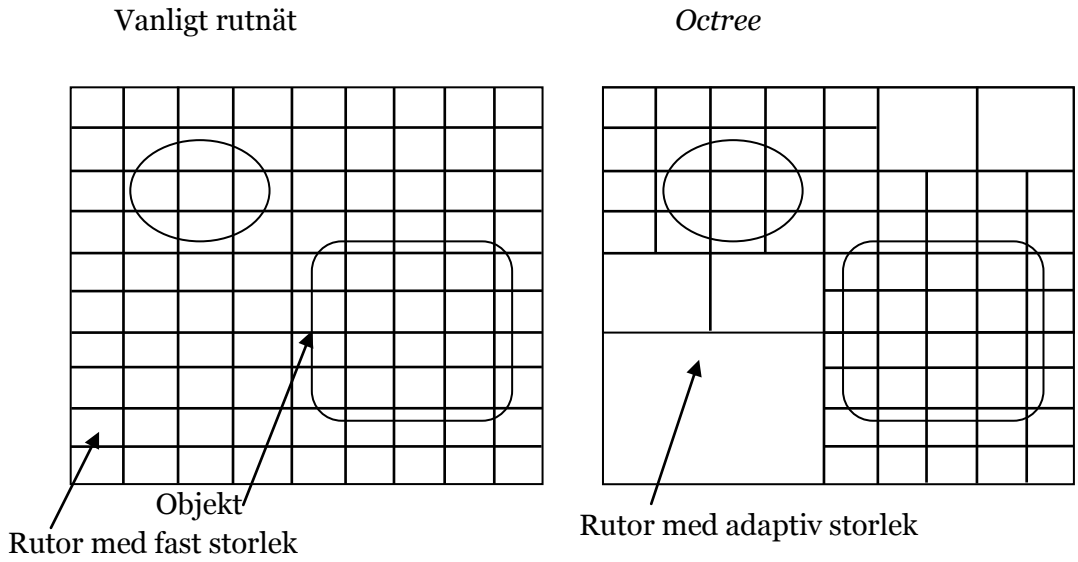
2.3 Rumsindelningsstrukturer

För att snabbt kunna svara på positionsförfrågningar i scenen och kunna möjliggöra olika tekniker för synlighet så säger Pantazopoulos och Tzafestas (2002) att utvecklaren av programmet bör dela upp scenen i spatiala datastrukturer. De presenterar fem av de vanligaste strukturerna som är ett vanligt rutnät, *octrees*, BSP-träd, kD-träd och *bounding volume* hierarki. Anledningen till att de är de mest kända strukturerna är att de är relativt effektiva och lätta att implementera. Problem som kan uppstå med de här strukturerna är bland annat om ett objekt ligger i två regioner samtidigt, skall utvecklaren då lägga objektet i båda regionerna eller duplicera objektet.

Pantazopoulos och Tzafestas (2002) skriver att den vanliga rutnätsstrukturen har rutor där storleken och antalet rutor är bestämda i förväg vilket gör att den strukturen inte kan anpassa sig till scenen och gör att den inte är den mest användbara.

Octrees bygger vidare på det vanliga rutnätet. I början är alla objekt bundna av en region som rekursivt delas in i åtta mindre regioner där de lagras och det upprepas tills strukturen

kommit till max djup eller att inga objekt finns i cellen och då har strukturen sina lövnoder. Det här leder till att vissa objekt lagras i mer än en lövnod. *Octrees* används också i *occlusion culling* algoritmer (Akenine-Möller & Haines, 2002). Meningen med den strukturen är att den är mer flexibel när det gäller densiteten och positionen för objekt i scenen. Figur 4 visar på hur skillnaden mellan vanliga rutnät och *octrees* kan se ut.

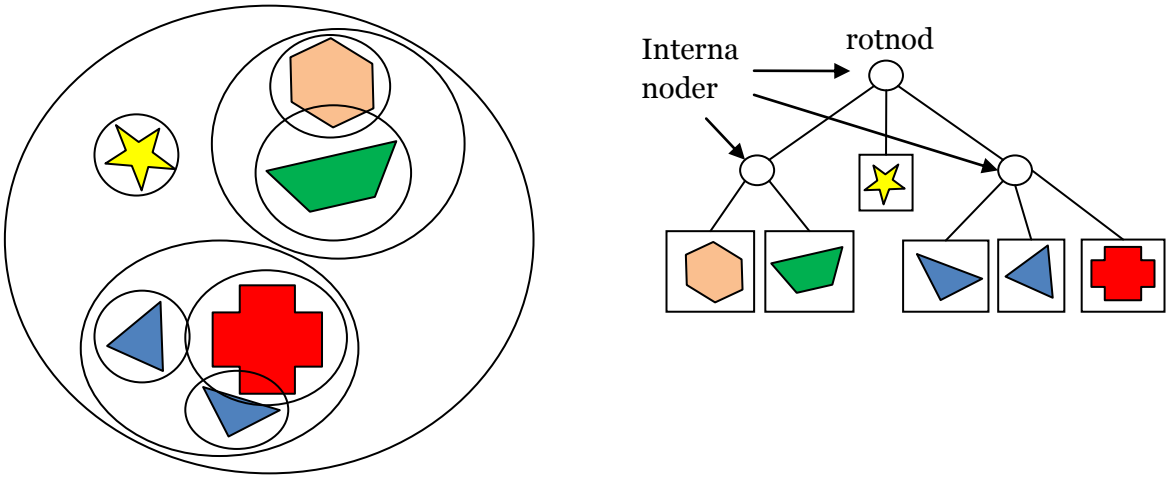


Figur 4 Vanligt rutnät- och *octree*-strukturerna (efter Pantazopoulos & Tzafestas, 2002).

Binary Space Partitioning-träd, förkortas BSP-träd, är en annan rumsindelningsalgoritm. Först väljs en polygon ut för att dela rummet i två delar med hjälp av ett plan. Efter det läggs de andra polygonerna i rummet i någon av de två delmängderna och om någon polygon korsas av en linje så delas den polygonen in i två delar och en av varje del läggs i de olika delmängderna. Det här fortsätter rekursivt tills alla polygoner är i BSP-trädet. Det är enligt Akenine-Möller och Haines (2002) svårt att göra ett tidseffektivt BSP-träd och sådana träd beräknas oftast endast en gång för att användas senare i applikationen.

kD-träd är en variant på ett BSP-träd som bygger på x-, y- och z-axlarna för att dela in polygonerna i olika regioner. Algoritmen fortsätter att dela upp rummet tills de uppsatta kriterierna blir uppnådda så som att användaren har satt ett maxdjup på trädet. *Axis-aligned BSP tree* är vad Bittner, Havran och Slavík (1998) kallar strukturen och de använder sig av den strukturen för att den är flexibel och enkel att bygga upp och traversera.

Bounding Volume (BV) hierarkier är en rumsindelningsstruktur som enligt Pantazopoulos och Tzafestas (2002) är mer anpassat efter formen på objekt i scenen än de andra rumsindelningsstrukturerna. Ett skrivbords volym har rummets nod som anfader medan en lådas volym har skrivbordets nod som anfader. Dessa volymer kan vara sfärer eller boxar m.fl. och flera objekt i världen kan vara omslutna av andra volymer vilket betyder att de ligger längre ner i hierarkin. Manocha, Salomon och Yoon (2003) använder sig av *oriented bounding boxes*, *OBBs*, för att få en volym bättre anpassad till objekten i scenen. *OBBs* kräver att deras algoritm gör mer beräkningar än om deras algoritm skulle använt sig av sfärer eller *axis aligned bounding boxes, AABBs,* men deras algoritm strukturerar upp allt under förbehandling av scenen. Figur 5 visar hur BV hierarkier är uppbyggda.



Figur 5 En BV hierarkisk struktur (efter Akenine-Möller & Haines, 2002).

Dessa är några av de vanliga rumsindelningsstrukturerna som används för att möjliggöra de *culling*-tekniker som introducerades i förra delkapitlet.

2.4 Exempel på *occlusion culling* i praktiken

Unity (2017) är en spelmotor som har inbyggt stöd för *occlusion culling*. Företaget som skapat systemet som Unity använder sig av är Umbra (2017). Det är många spelföretag som använder sig av Umbras system för bland annat *occlusion culling*. I spelet Doom som släpptes 2016 så finns Umbras system för *occlusion culling* och Courrèges (2016) skriver att det används i samband med spelets egna *occlusion*-förfrågningar.

I manualen för Unity (2017) så finns det information om hur utvecklare skall gå till väga för att ha med *occlusion culling* i spelet som utvecklas och det nämns även att spelen får förbättrad prestanda då mindre anrop till motorns utritningsmetod utförs. Sättet utvecklare går till väga för att använda sig av *occlusion culling* i Unity är genom att sätta objekt till att antingen vara statiska *occluders* och/eller *occludees*. *Occludees* är objekt som kan döljas av *occluders* men döljer inte själv andra objekt. Det utvecklare måste tänka på är att om objekt är små eller transparenta är de inte lämpade som *occluders*. Markerar utvecklaren felaktigt objekt som *occluders* utför Unity onödiga beräkningar på dessa objekt. Unity delar också in scenen i celler som sparas i BSP träd. Två olika träd sparas för statiska och dynamiska objekt och används för att snabba upp *occlusion*-förfrågningar. När en utvecklare har bestämt alla inställningar kan *occlusion* data genereras och sparas ner som sedan används av systemet vid körning.

3 Problemformulering

Det är viktigt att på ett effektivt sätt kunna identifiera polygoner som ska ritas ut och avfärda de osynliga polygonerna tidigt i grafikpipelinen (Coorg & Teller, 1997). På så sätt görs inga onödiga ljusberäkningar, texturmappning och liknande operationer på de dolda polygonerna och då sparas beräkningskraft. Att välja en delmängd av alla synliga objekt att använda som *occluders* förbättrar generellt sätt prestandan. Därför är det intressant att undersöka olika mängder med *occluders* i simuleringar och hur annorlunda resultatet blir med olika mängder. Tidigare projekt använder sina algoritmer i scener som de satt upp själva eller som de lånat från andra projekt. Därför är det också intressant att se hur *occlusion culling* presterar i scener som kommer från olika typer av spel där kameraåkningar och scener skiljer sig från varandra. Typiska scener i spel behöver inte nödvändigtvis ha objekt som uppenbart kan användas som *occluders* vilket gör att tester i sådana scener kan vara värdefullt att utföra.

3.1 Tidigare projekt 1: *Occlusion culling* för modeller med stora *occluders*

Coorg och Teller (1997) gjorde en implementation av *occlusion culling* som fokuserar på att använda sig av stora *occluders*. I deras rapport kom de fram till en effektiv algoritm som gör att de kan skära bort en stor del av objekten i scenen så att de inte behöver ritas ut. Genom att använda sig av ett kD-träd för objekten i scenen kan de effektivt beräkna vilka delar som är synliga. De bygger upp trädet i ett förbehandlingssteg. De utnyttjar spatial och temporal koherens så att de inte hela tiden behöver bygga upp relationerna om vad som är dolt från grunden utan kan enklare återanvända de undansparade relationerna för att snabba upp förfrågningarna. De använder en liten delmängd av objekten i scenen som *occluders* vilket de motiverar med att det oftast är en få objekt som döljer större delen av geometrin i scenen. Vilka objekt som används som *occluders* sker dynamiskt när kameran ändrar position i scenen och beror på en estimering av rymdvinkeln.

I projektet så skriver Coorg och Teller (1997) att deras algoritm använder sig av plan i den tredimensionella rymden för att upptäcka vilka objekt som är dolda vilket betyder att deras algoritm är en objektrumsalgoritm. Krav på att *occluders* ska vara konvexa finns också i deras algoritm för att undvika problem som kan uppstå med uträkningen av dolda objekt.

Testerna som de utförde var att göra körningar i två olika scener där det fanns flera tusen objekt som kunde användas som *occluders*. Resultaten av deras experiment var att *occlusion culling* minskade renderingsbelastningen med en faktor av 6 till 8 än om de bara använde *frustum culling* när de gjorde sina tester. De sparar undan hur lång tid det tar att köra *culling* algoritmerna och hur lång tid det tar att rendera scenen. De uttrycker också att det hade varit intressant att undersöka scener som inte har stora *occluders* samt att undersöka hur det ser ut när algoritmen använder sig av olika antal *occluders* i sina beräkningar.

3.2 Tidigare projekt 2: Hierarkisk *culling* med *occlusion trees*

Bittner, Havran och Slavík (1998) är också några som implementerat *occlusion culling* med hjälp av *occlusion trees* vilket är ett BSP träd som baseras på en mängd av *occluders* och en utsiktspunkt. De har även ett modifierat BSP träd för att visa på att deras modifierade träd kan förbättra prestandan. De har två olika algoritmer för att bestämma synligheten för en polyeder. Att använda sig av ett BSP träd gav dem en exakt algoritm när det gällde vad som

var synligt. Genom att använda deras modifierade BSP hade de också en konservativ algoritm för vad som var synligt.

De använder sig inte av någon förbehandling för att välja ut *occluders* utan markerar endast objekt som kan användas som *occluders* vilket betyder att de behöver veta om hur scenen ser ut i förväg. Sedan använder de sig av alla potentiella *occluders* för att dynamiskt välja ut *occluders* under körning. För att bestämma värdet av potentiella *occluders* använder de en estimering av rymdvinkeln som gåtts igenom i tidigare kapitel. De bestämmer att alla *occluders* måste vara konvexa så att de kan bygga upp skuggvolymer som de bygger upp av plan vilket beräknas genom en utsiktspunkt och en *occluder*. Genom att räkna ut vad som ligger i skuggregionerna kan de bestämma de regioner som är synliga, delvis synliga och inte synliga.

De använde sig av genomgångar i olika inomhusmiljöer för att mäta sina algoritmer och där så var det alltid tidseffektivare när de använde sig av sina olika algoritmer jämfört mot när de bara körde med *view frustum culling*. De testade även att använda sig av olika antal *occluders*. De summerade sina testfall och hur mycket tid som gick åt för att utföra de olika algoritmerna samt tiden för varje bildruta i scenen. Resultatet sparades ner för de olika algoritmerna som använde ett varierande antal *occluders*. Hur många polygoner som renderades sparades också ner.

3.3 Tidigare projekt 3: Accelerated *occlusion culling* using shadow frusta

Cohen et al. (1997) skrev i sitt projekt om att *occlusion culling* kan delas in i två delproblem. Det första är att hitta en liten delmängd av rimliga *occluders* som kan användas. Med hjälp av delmängden kan det andra problemet lösas genom att den osynliga geometrin markeras så att den inte ritas ut. De två delproblemen är delvis oberoende av varandra och gås igenom i detalj i deras projekt.

För att välja värdefulla *occluders* använder Cohen et al. (1997) sig av den estimerade rymdvinkeln på potentiella *occluders* för att få en uppfattning hur mycket geometri de kan dölja från en utsiktspunkt. Dessa potentiella *occluders* sparas ner i en spatial datastruktur i ett förbehandlingssteg som sedan används under deras körningar. Alla andra objekt i scenen sparas i en separat datastruktur. I deras förbehandlingssteg sparar de ner genomsnittet på hur många objekt en *occluder* döljer från olika regioner i scenen. De använder den informationen för att veta hur värdefulla potentiella *occluders* är i olika regioner och använder den informationen för att spara *occluders* som ger bra resultat. Under körning väljer de ut ett antal aktiva *occluders* som används till *occlusion*-förfrågningar. För att optimera sin algoritm så utnyttjar de temporal koherens genom att sortera listan av *occluders* efter de som dolde mest i förgående uppdatering.

Deras *occlusion culling*-algoritm består av att bygga upp en *occlusion*-volym som är skuggan från alla potentiella *occluders* och sedan gör de tester för att se om objekt helt omsluts av den volymen vilket betyder att de är dolda. Sättet de bygger upp volymerna är att skapa plan från kanter på konvexa *occluders* i scenen.

Sättet de mäter sin algoritm är att använda den i en stadsscen. De mäter hur lång tid varje uppdatering tar med bara *view frustum culling* jämfört med sin *occlusion culling*-algoritm. De skriver också ut genomsnittet på hur många polygoner som renderas per uppdatering.

Genomsnittet av dolda objekt ökar inte avsevärt desto fler *occluders* de använder i scenen när algoritmen använder i snitt 8 *occluders* per uppdatering.

3.4 Syfte och delmål

Syftet med det här projektet är att se hur *occlusion culling* och hur valet av *occluders* kan förbättra prestandan i olika typer av vanliga scener från spel. Tidigare tester visar på att användandet av tekniken har lett till att scener renderas snabbare och det totala antalet polygoner som ritats ut reduceras. Eftersom tidigare projekt inte fokuserat på scener från spel i sina tester så är det intressant att se hur väl *occlusion culling* fungerar i just sådana scener. Det är även intressant att se hur valet av antalet *occluders* som används påverkar hur mycket geometri som ritas ut samt vad det blir för overhead på algoritmen när antalet varieras.

3.4.1 Delmål 1: scener och *occlusion culling*-algoritm

Det första delmålet i projektet är att skapa olika scener som är baserade på olika typer av spel. I de här scenerna ska typiska kameraåkningar för just den speltypen sättas upp så att det efterliknar hur det vanligtvis ser ut i spelet. Det krävs att en algoritm för *occlusion culling* implementeras som kan användas i de här scenerna. Det skall gå att bestämma hur många *occluders* som skall användas vid varje körning.

Efter en körning har genomförts så skall information från körningen sparas ned. Det som behövs är renderingstiderna, overheaden av att köra *occlusion culling*-algoritmen och antalet polygoner som ritas ut i varje uppdatering.

3.4.2 Delmål 2: utvärdering

Det här delmålet går ut på att utvärdera körningar i scenerna som satts upp för att se hur väl *occlusion culling* fungerar i dem. Scenerna ska köras med olika antal *occluders* så att det ger en förståelse av effekten med att ha ett varierande antal med *occluders* genom att sammanställa resultatet från körningarna och jämföra med varandra. Det här projektet ska undersöka om det är stora skillnader i tiderna för rendering och overheaden med *occlusion culling*-algoritmen. Det ska också undersöka hur mycket geometri som inte renderas ut när algoritmen kör med ett varierat antal med *occluders*.

3.5 Metodbeskrivning

Projektet har möjliggjorts genom att använda sig av metoder som liknar de som finns i tidigare projekt. Den första metoden som presenteras handlar om hur scenerna har satts upp och algoritmen som implementerats för *occlusion culling*. Den andra metoden är själva experimentet i de scener som satts upp och mätningen av värdena som givits av körningarna.

3.5.1 Metod för delmål 1: Scener och *occlusion culling*-algoritm

För att realisera det första delmålet så implementerades en objektrumsalgoritm för *occlusion culling* i Unity. I algoritmen finns det möjlighet att välja hur stor mängd av *occluders* som ska användas i en körning. *Occluders* sorteras i storleksordning när de väljs ut. För att kunna utvärdera algoritmen så lades det till tidtagning för hur lång tid det tar att köra *occlusion culling*-algoritmen samt hur lång tid det tar att rendera en scen. Det lades till stöd för att räkna ut totalt antal polygoner i en scen och hur många som ritas ut.

Det skapades tre olika scener som efterliknar olika speltyper. De scener som skapats är Dust 2 från Counter-Strike spelserien av Valve Corporation (2012), Peach's Castle från Super Mario 64 av Nintendo (1996) och Lost Temple från Starcraft 2 av Blizzard Entertainment

(2010). De här scenerna sattes upp i Unity. I dessa scener så skapades automatiserade kameraåkningar som beter sig som kamerorna vanligtvis gör för dessa spel så det ges en överblick hur väl *occlusion culling*-algoritmen kan prestera för de olika spelen. Anledning till att just de här spelen valdes var för att de skiljer sig i hur scenerna normalt är uppbyggda samt till att kameran i spelen skiljer sig åt så det ger en förståelse för hur väl *occlusion culling* kan prestera i liknande spel.

3.5.2 Metod för delmål 2: experiment

I det andra steget så utvärderades de scener som satts upp genom att göra tre körningar per scen med 5%, 10% och 20% av alla *occluders*. Eftersom det satts upp automatiserade kameraåkningar så var förutsättningarna likvärdiga för varje körning och scenerna förändrades inte på något sätt. Overheaden för att köra *occlusion culling* förväntades vara större när fler *occluders* användes. Resultaten av körningarna infördes i tabeller och grafer för att få en överblick av tiden det tar att rendera scenen, overheaden av algoritmen och antalet renderade polygoner så det ger en bild av hur scenerna påverkades av det här projektets *occlusion culling*.

Scenen som baseras på Dust 2 förväntades att påverkas mest av olika antal *occluders* då den scenen är uppdelad i naturliga regioner med flera *occluders* som skiljer dem åt. Scenen borde också vara den som får mest värde av att använda sig av *occlusion culling*-algoritmen jämfört med de andra scenerna.

Den scen som är baserad på Peach's Castle förväntades att inte påverkas så mycket när körningar med olika antal med *occluders* genomfördes. Eftersom det är en öppen scen finns det få objekt som döljer mycket geometri och endast från ett fåtal utsiktspunkter där det är många objekt som kan döljas.

Scenen som liknar Lost Temple förväntades att inte ha nån större skillnad med olika antal *occluders* då kameran alltid ser ner över scenen där det knappt döljs några objekt. Det borde alltid vara rätt jämt mellan körningarna i hur många objekt som renderas.

Den *occlusion culling* som finns inbyggt i Unity användes också i en körning per scen med de standardvärden som Unity ger. En körning utan någon *occlusion culling* kördes också och då används den *view frustum culling* som alltid är påslagen i Unity. I de körningarna så mättes endast overheaden för *culling* och hur lång tid det tar att rendera scenen då möjligheten att räkna antalet renderade polygoner inte fanns tillgänglig.

3.5.3 Diskussion kring val av metod

I det här projektet valdes det att utföra experimenten med de ovannämnda metoderna för att försöka efterlikna de tidigare projektens metoder. På så sätt kan resultat som ges i det här projektet enklare jämföras mot resultaten från deras projekt. Det är enkelt att upptäcka fel i det här projektet om värden som fås från experimenten inte ligger i närheten av de förväntade resultaten. Att utvärdera *occlusion culling* i olika speltyper kan ge insikt i när den tekniken är lämplig att använda.

Det är intressant att se hur väl algoritmen i det här projektet presterar i jämförelse med den algoritm som finns i Unity. Då kan utvecklare se om det är värt att lägga tid på att utveckla sina egna algoritmer eller om det är lika bra att använda sig av Unitys algoritm.

4 Genomförande

I detta kapitel görs en genomgång av implementationen som valts för det här arbetet och avvägningar för valet av algoritmen. Den mest komplicerade beskrivningen är den om *occlusion culling*-algoritmen och den valda lösningen där. Det finns en jämförelse över hur en scen ser ut med och utan *occlusion culling*. Arbetet i Unity beskrivs och hur resultatet av körningarna samlats in. Sist i kapitlet finns det en genomgång av de scenerna som satts upp och valet av att använda dem i det här projektet.

4.1 Val av algoritm

I det här projektet så implementerades en *occlusion* culling-algoritm som gör beräkningar i objektrummet. En anledning till det valet var att studier utfördes på tidigare projekt som implementerat objektrumsalgoritmer och de kunde visa på stora förbättringar i körningar i sina scener. Coorg & Teller (1997) gjorde en objektrumsalgoritm som de använde i två olika scener där det förbättrade renderingstiderna. Bittner, Havran och Slavík (1998) gjorde också en objektrumsalgoritm som de använde i olika inomhusscener för sina mätningar. Cohen et al. (1997) har också gjort en objektrumsalgoritm som fungerar likt de andra och använder den i en stadsscen där de utförde olika mätningar.

En annan anledning till att det implementerades en objektrumsalgoritm var att de bildrumsalgoritmer som studerades var komplicerade och svåra att förstå. Att implementera en bildrumsalgoritm hade tagit lång tid och projektet hade inte hunnit bli klart. Tiden det tog att implementera algoritmen var över förväntningarna och därför implementerades endast en algoritm.

Att dela upp scenerna i regioner med hjälp av en rumsindelningsstruktur hade varit bra för att optimera algoritmen som flera tidigare projekt visat. Att ha flera objekt del av en region gör att algoritmen kan eliminera hela regionen om den är dold istället för att kontrollera varje objekt i regionen. Det tog dock så lång tid att utveckla algoritmen och att få den att fungera med enklare *bounding volumes* så då gjordes valet att inte implementera en rumsindelningsstruktur.

4.2 *Occlusion culling*-algoritm

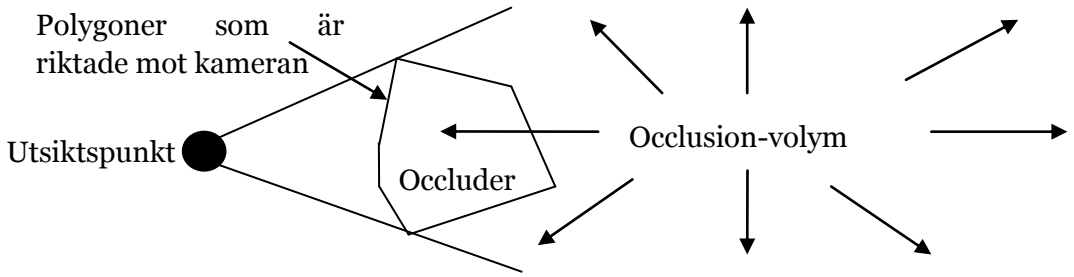
En algoritm för *occlusion culling* har implementerats. Det är möjligt att välja hur många *occluders* från en scen som ska användas av algoritmen. En anledning till det är att om alla objekt används som *occluders* så får algoritmen en tidskomplexitet på $O(n^2)$ där n är antal objekt i scenen. Att undersöka hur algoritmen påverkas av olika antal med *occluders* är en annan anledning. Bittner, Havran och Slavík (1998) utförde experiment med olika antal av *occluders* för att se påverkan på deras körningar. Cohen et al. (1997) lägger också stor vikt på valet av *occluders* och dess inverkan på resultaten. De beskriver att när de ökar antalet *occluders* bortom en viss gräns så ses inte stora förbättringar i deras tester. Aila & Miettinen (2004) beskriver hur de tar en delmängd av synliga objekt som *occluders* för att öka prestandan.

För att kunna undersöka hur effektiv algoritmen är så har stöd lagts till för att mäta tiden det tar att utföra beräkningarna samt hur många polygoner som ritas ut.

4.2.1 Skapa *occlusion*-volymer

Sättet som Cohen et al. (1997), Bittner, Havran och Slavík (1998) och Bacik (2002) utför sina *occlusion*-förfrågningar är att de bygger en *occlusion*-volym och alla objekt som ligger i den

volymen är då dolda. Figur 6 visar hur en sådan volym kan se ut. I deras projekt så är alla *occluders* konvexa objekt för att planen som används inte ska returnera fel information om ett objekt är dolt eller inte. I det här projektet så används enkla *axis aligned bounding boxes*, *AABBs*, för att bygga upp *occlusion*-volymer. Anledningen till det var att det blev komplext nog att implementera en lösning som fungerar dynamiskt med flera objekt i scenen. I scenerna är det endast kuber som markeras som potentiella *occluders* för att sfärer med dess omslutande *AABB* skulle ge en felaktig *occlusion*-volym.



Figur 6 En representation av en *occlusion*-volym (efter Bacik, 2002)

I ett förbehandlingssteg så sparas det för varje *occluder* dess polygonytor med polygonytornas tillhörande kanter när användaren startar programmet. Det här uppdateras inte under körning då alla *occluders* är statiska. Det som uppdateras under körning är vilka kanter och polygonytor som är synliga samt avståndet till kameran vilket används för beräkningen av *occlusion*-volymen. Om en potentiell *occluder* inte blir vald i förbehandlingssteget så markeras den att istället vara en *occludee* i körningen.

I det här projektet gjordes beslutet att inte ha objekt som rör på sig under körning eftersom alla objekt har en lista med hörn som används för att göra kontroller om det är synligt eller inte. Om det funnits en rumsindelningsstruktur hade objekt som rör på sig behövt hanteras separat. Att uppdatera positionen på objektet skulle inte förändra själva algoritmen även om resultatet av körningarna påverkats.

4.2.2 *Occlusion culling* med undansparade relationer

Occlusion-algoritmen består av flera viktiga delar. Eftersom det är en fördel att utföra så få jämförelser som möjligt sparas alla relationer om vilken *occluder* som döljer vilken *occludee* i en hashtabell. Om en *occludee* inte är dold av någon *occluder* kommer det ske jämförelse mot samtliga synliga *occluders*.

För att möjliggöra uppbyggandet av *occlusion*-volymer så används funktionen *getVisibleOccluderPart* som beskrivs i figur 7. Genom den hämtar algoritmen de kanter och polygonytor i form av plan som används när algoritmen senare bygger upp alla de plan som utgör en *occlusion*-volym i *updateOcclusionVolume* som beskrivs i figur 8.

```
function getVisibleOccluderPart(Occluder o)
	b = boxens plan med tillhörande kanter som tillhör o
	om(b == NULL)
		return
	töm listan o.synligaKanter
	töm listan o.synligaPlan
	för alla plan med tillhörande kanter i b
		om(planets avstånd från origo + skalärprodukten av planets normal
```

```
och kamerans position >= 0)
                    för alla kanter för planet
                          om(kanten finns i o.synligaKanter)
                                ta bort kanten
                          annars
                                lägg till kanten
                    lägg till planet
```

Figur 7 Pseudokod för när algoritmen uppdaterar en *occluder*s synliga polygoner och kanter.

```
function updateOcclusionVolume(Occluder o)
      töm listan o.occlusionFrustum
      för alla sidor i o.synligaKanter
            ny normal n
            n = kryssprodukten av kantens första hörn - kamerans position och
kantens andra hörn - kamerans position
            lägg till nytt plan i o.occlusionFrustum med normaliserad normal n
och positionen kantens första hörn
      för alla plan i o.synligaPlan
            lägg till plan i o.occlusionFrustum
```

Figur 8 Pseudokod för när algoritmen uppdaterar en *occluders occlusion*-volym.

I *updateVisibleOccluders* markeras de *occluders* som ska användas för *occlusion*-förfrågningarna. Funktionen kontrollerar om en *occluder* ligger i kamerans *view frustum*. Den kontrollerar även om *occludern* döljs av andra *occluders* för att inte använda redan dolda *occluders*. Om en *occluder* som var synlig i förra uppdateringen är dold uppdateras relationstabellen och *occluderns* relaterade *occludees* plockas bort. Det här beskrivs i detalj i figur 9.

```
function updateVisibleOccluders()
      töm listan visibleOccluders
      för alla occluders
            om(occludern är i kamerans view frustum)
                  dold = false
                  för alla synliga occluders
                        om(occluder döljs av synlig occluder)
                              occluder.renderer = false
                              dold = true
                              break
                  om(dold == false)
                        occluder.renderer = true
                        getVisibleOccluderPart(occluder)
                        updateOcclusionVolume(occluder)
                        lägg till occluder i visibleOccluders

      töm listan toRemove
      för alla occludee i occludedBy.Keys
            om(visibleOccluders inte innehåller occludedBy.GetValue(occludee))
                  lägg till occludee i toRemove
                  occludee.renderer = true
      för alla occludee i toRemove
            ta bort occludee från occludedBy
```

Figur 9 Pseudokod för att uppdatera synliga *occluders*.

För att ta reda på vilka *occludees* som är potentiellt synliga kontrollerar algoritmen om de är i kamerans *view frustum*. Är de det läggs de till i en lista över potentiellt synliga *occludees*.

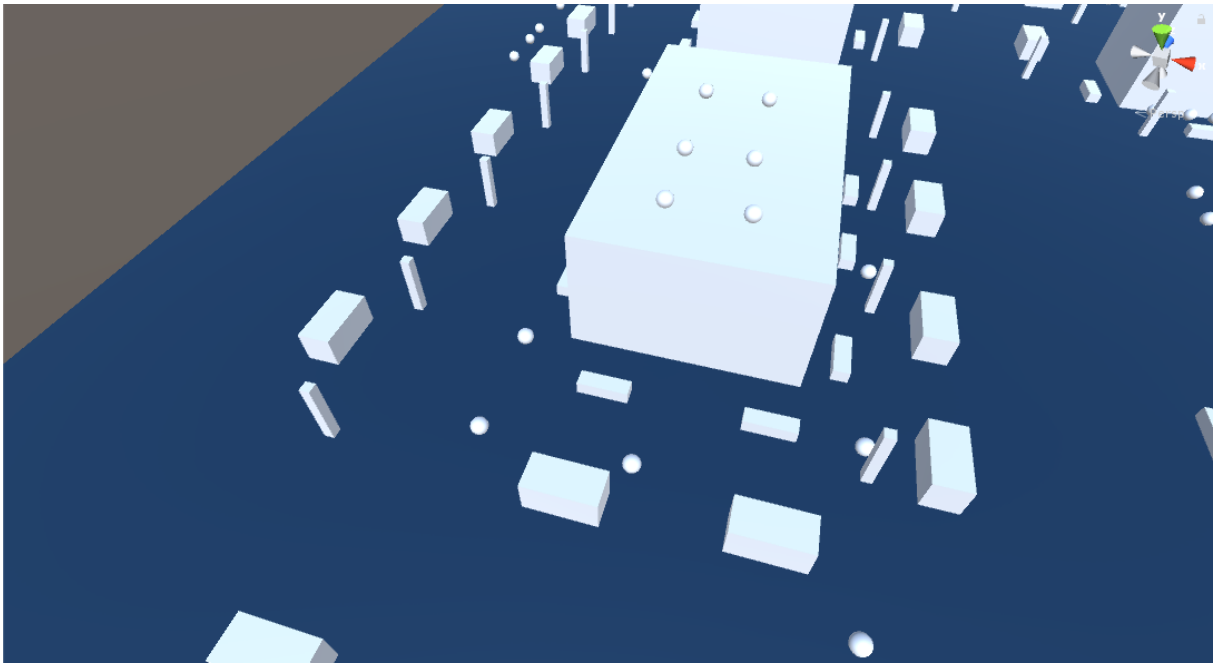
Annars görs en kontroll för att se om de var dolda i förra uppdatering. Var de dolda i förra uppdateringen och inte är i kamerans *view frustum* tas de bort från relationstabellen då det inte behöver ske ytterligare kontroller för att se om de är synliga eller inte. Om de inte är potentiellt synliga och inte var dolda i förra uppdateringen så flaggas de att inte renderas.

När all information om vilka *occluders* som skall användas och vilka *occludees* som potentiellt är synliga samlats ihop utförs algoritmens sista steg. För alla potentiellt synliga *occludees* görs en kontroll om de är dolda eller inte. Om en *occludee* var dold i förra uppdateringen kontrolleras endast den *occludern* som dolde *occludeen*. Är objektet inte längre dolt så tas den bort från relationstabellen och flaggas att den ska renderas. Annars görs kontroller mot alla *occluders* och är objektet dolt så flaggas den att den inte ska renderas samt så läggs relationen om vilken *occluder* som dolde den till i relationstabellen. Är objektet inte dolt så flaggas det att det ska renderas. Funktionen *IsOccluded* är rättfram och beskrivs i figur 10.

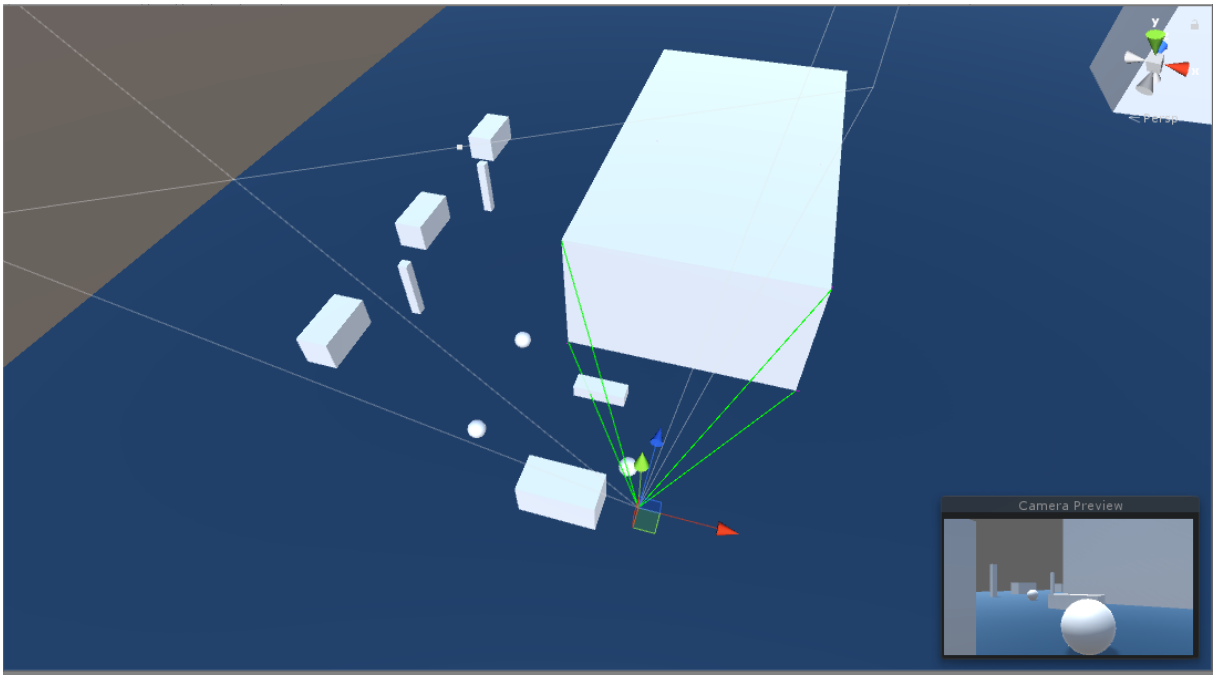
```
function IsOccluded(lista av plan p, lista av hörn)
	för alla hörn
		för alla plan
			om(planets avstånd från origo + skalärprodukten av planets
normal och hörn >= 0
				return false
	return true
```

Figur 10 Pseudokod för att bedöma om ett objekt är dolt.

I figur 11 så kan en testscen beskådas utan några beräkningar som kan jämföras mot figur 12 som visar användningen av *occlusion culling*.



Figur 11 Skärmdump från en scen utan *occlusion culling*.



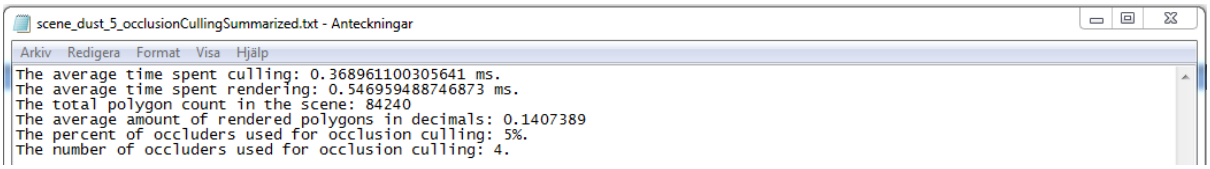
Figur 12 Skärmdump från en scen med *occlusion culling* igång.

4.3 Implementation i Unity spelmotor

För att kunna sätta upp scener på ett enkelt sätt så bestämdes det att implementera lösningen i Unity spelmotor med C#-skript. På så sätt var det enkelt lägga till objekt och bygga upp lösningen från grunden istället för att försöka lägga till det i ett tidigare spelprojekt som författaren av det här arbetet tagit del av där tanken på att ha stöd för *culling*-algoritmer inte fanns. Att bygga upp allt från grunden i OpenGL hade också tagit lång tid för att nå samma resultat som åstadkommits med hjälp av Unity.

I Unitys editor kan utvecklaren lägga till objekt i scener. Till dessa objekt kan utvecklaren lägga till komponenter vilket kan vara saker som renderingskomponent, kamerakomponent, kolliderarkomponent eller eget skrivna skript. Genom att utnyttja detta systemet kan utvecklaren snabbt stänga av och på renderingskomponenter samt de skript utvecklaren skrivit. För att alla testfall ska ha samma förutsättningar är det möjligt att animera en kameraåkning som kan användas i scenerna.

För att kunna utvärdera algoritmerna så valdes det att mäta hur lång tid det tar för *culling*-algoritmerna att köra. Alla värden sparas för att sammanställas i en lista som skrivs ut när körningen är klar. Samma sammanställning sker för renderingstiderna som sparas i en egen fil. Antal renderade polygoner i varje bild sparas också till fil. Slutligen så summeras körningen där det som sparas är den genomsnittliga tiden att köra *culling*-algoritmen, den genomsnittliga renderingstiden, hur många polygoner det finns totalt i scenen och hur många polygoner som renderas i snitt i procent. Det sparas även hur många *occluders* som användes i körningen och hur stor procent av alla potentiella *occluders* som valdes. Figur 13 visar exempel på hur filen kan se ut efter att programmet har körts.



Figur 13 Skärmdump av summeringen från en körning.

De körningar som sker med och utan Unitys *occlusion culling*-algoritm sparar endast overheaden det tar att utföra *culling* och tiden det tar att rendera scenen. Anledningen är att Unity inte har ett API för att hämta antalet renderade polygoner. I det här projektets *occlusion culling*-algoritm så räknas antalet renderade polygoner manuellt. Det fanns inte tid att försöka lägga till den funktionaliteten för Unitys körningar.

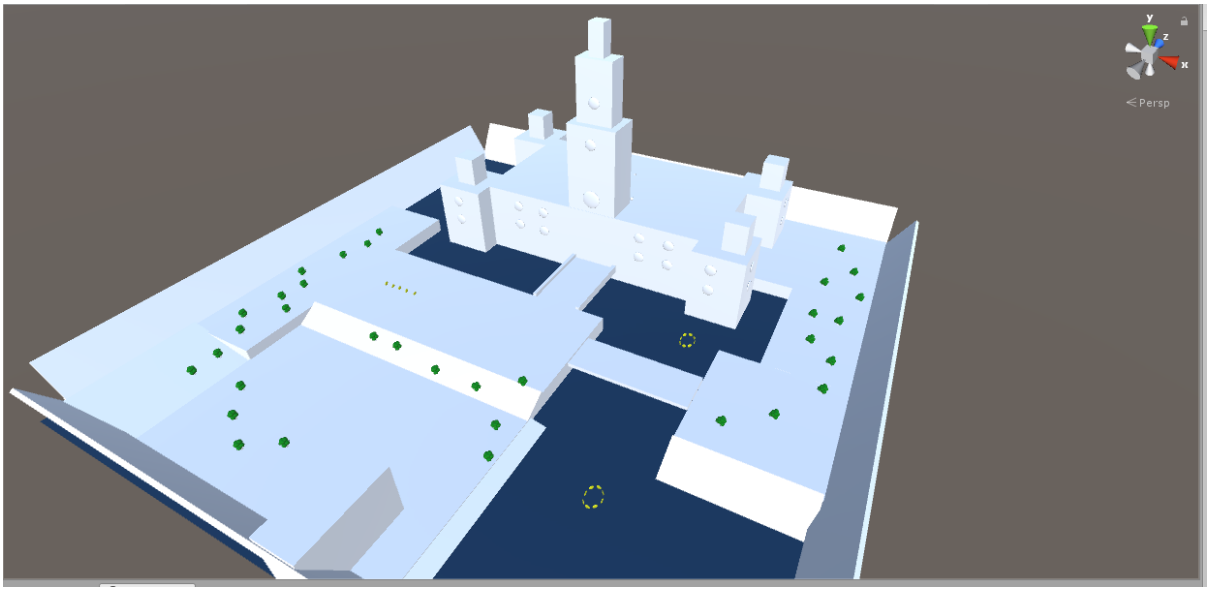
4.3.1 Scenen Dust 2

Scenen som baseras på Dust 2 visas i figur 14. Det som är intressant med den scenen är att den är en lämplig kandidat för *occlusion culling*. Tidigare projekt har fokuserat mycket på inomhusmiljöer eller scener med stora *occluders* och i den här scenen så finns det lite av båda. Kameraåkningen sker strax ovanför marknivå genom scenen. Den är återskapad för att efterlikna hur det vanligtvis ser ut i spelserien Counter-Strike.

Figur 14 Överblick av scenen Dust 2 i Unity.

4.3.2 Scenen Peach's Castle

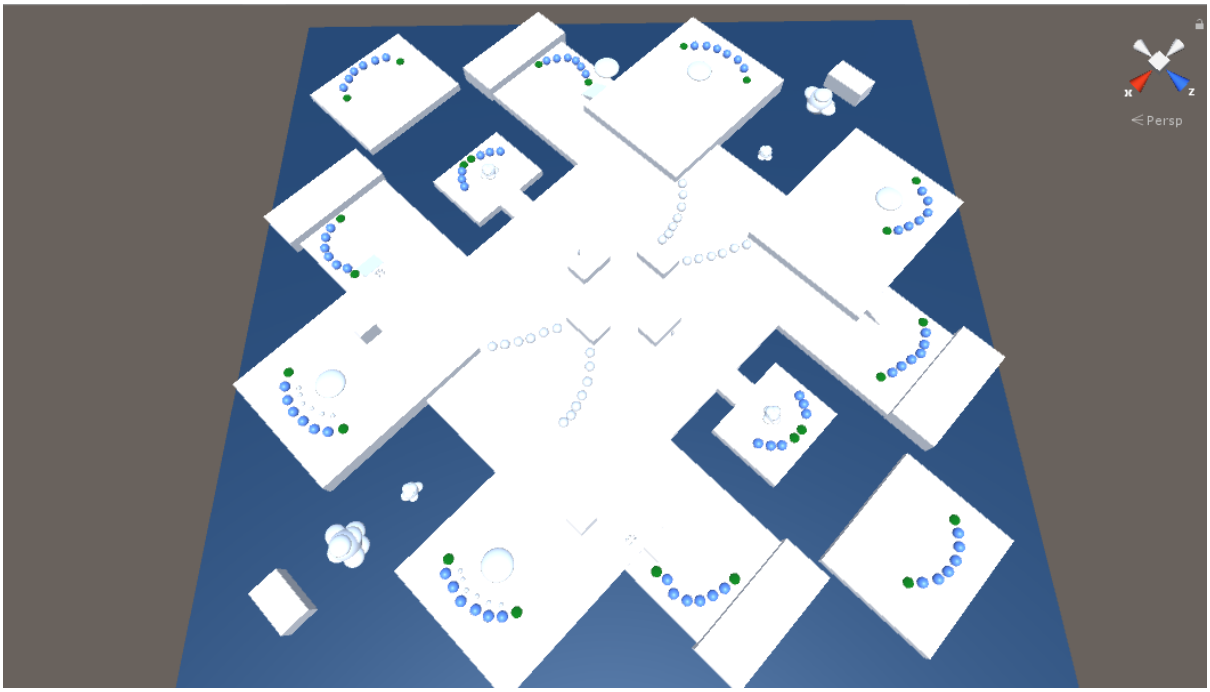
Den här scenen baseras på öppningssekvensen av Super Mario 64 och kan beskådas i figur 15. I det spelet så fokuserar kameran på spelaren och följer efter den vanligtvis en bit över marken. Det intressanta med den här scenen är att det är mycket mindre naturliga *occluders* i jämförelse med Dust 2. Det finns en del stora *occluders* men de är inte utplacerade så att de döljer många objekt.



Figur 15 Överblick av scenen Peach's Castle i Unity.

4.3.3 Scenen Lost Temple

Den scen som är minst lämpad för *occlusion culling* är scenen Lost Temple från Starcraft 2 och det här projektets tolkning kan beskådas i figur 16. I det här spelet så är kameran vanligtvis i luften och blickar ner mot marken. Eftersom kameran alltid har en överblick ovanifrån så är det få objekt som döljs. Det är därför intressant att se om det finns något värde i att använda *occlusion culling* i en sådan scen.



Figur 16 Överblick av scenen Lost Temple i Unity.

5 Resultat

I det här kapitlet finns resultaten av de utförda experimenten. Först så summeras körningarna i de olika scenerna och effekten av att använda en varierande mängd *occluders* i de scener som har satts upp. Antalet utritade polygoner visas också för varje körning för att få grepp om hur valet av *occluders* påverkar körningarna. Eftersom det inte var möjligt att få ut samma information i körningarna gjorda med Unitys algoritmer så presenteras det resultatet separat. Slutligen så analyseras resultatet av experimentet och så beskrivs de slutsatser som har gjorts.

5.1 Presentation av undersökning

För att få en förståelse för hur väl algoritmen ter sig för olika speltyper har algoritmen använts i scener som skiljer sig från varandra i hur de är uppbyggda och hur kameraåkningar vanligtvis ser ut för de speltyperna. Tidigare projekt har ofta fokuserat på scener där *occlusion culling* förväntades ge ett bra resultat. I det här projektet har scenerna varierats för att simulera hur olika speltyper är uppbyggda. Undersökningen i det här projektet utfördes på ett sätt så att det enkelt går att jämföra med resultat från tidigare projekt.

För att få en överblick av resultatet så summeras körningarna från scenerna. Varje körning har utförts i totalt 3600 bildrutor där tiden för overheaden av algoritmen och tiden för rendering varierar. För att ge en tydlig bild hur algoritmen presterade under körningarna så presenteras den genomsnittliga tiden samt det minsta och högsta värdet för overheaden av algoritmen. Notera att de 10 första bildrutorna inte inkluderades när det högsta värdet valdes ut då det tar några bildrutor innan körningen har startat helt och hållet. Antalet *occluders* som valts ut av den totala mängden med potentiella *occluders* presenteras. Antalet varierar för varje scen då potentiella *occluders* skiljer sig mellan scenerna. Efter det så presenteras de renderingstider som gavs under körningarna. Även för renderingstiderna så användes inte de 10 första bildrutorna när det högsta värdet på renderingstiden valdes ut. Genomsnittet på hur många polygoner som renderades under körningen presenteras för att ge en överblick på hur mycket algoritmen påverkade en scen. Grafer för hur många polygoner som renderades i varje bildruta presenteras för att ge en bild av hur körningarna skiljer sig åt.

5.1.1 Scenen Dust 2

I det här delkapitlet presenteras resultatet av algoritmen för scenen Dust 2. Som det går att se i tabell 1 så ökar overheaden för algoritmen när fler *occluders* används. I tabell 2 går det att se att renderingstiden sjunker när fler *occluders* används och att antalet renderade polygoner reduceras också när fler *occluders* används. Det finns perioder i körningarna där lite eller ingen *culling* utförs vilket syns då den minsta tiden för rendering är nästan likadan.

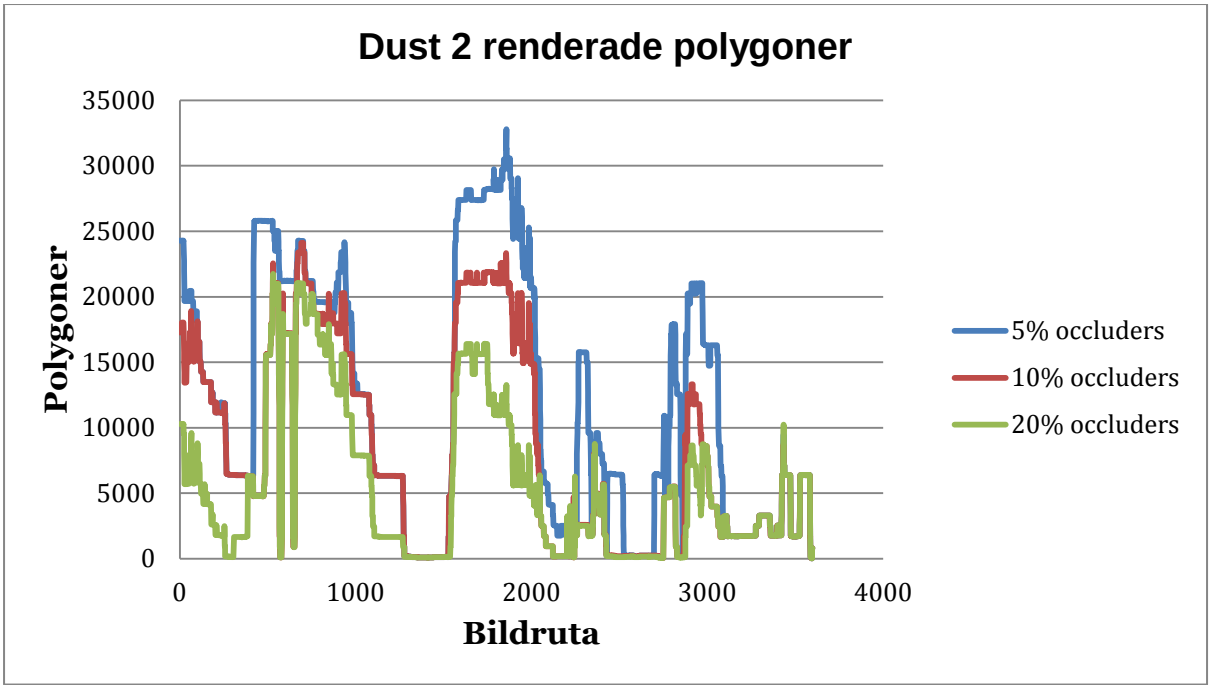
Tabell 1 Tabell som visar overhead av algoritmen i körningarna från scenen Dust 2.

Antal *occluders* i procent	**Antal utvalda *occluders***	**Genomsnittlig overhead [ms]**	**Minsta overhead [ms]**	**Högsta overhead [ms]**
5%	4	0.369	0.085	1.281
10%	9	0.449	0.095	1.558
20%	19	0.552	0.108	1.769

Tabell 2 Tabell som visar renderingstiderna i körningarna från scenen Dust 2.

Antal *occluders* i procent	**Genomsnittlig renderingstid [ms]**	**Minsta renderingstid [ms]**	**Högsta renderingstid [ms]**	**Renderade Polygoner %**
5%	0.547	0.192	1.084	14.07
10%	0.499	0.189	0.914	9.93
20%	0.461	0.193	0.891	6.66

Figur 17 visar hur många polygoner som ritades ut under tre körningar med olika mängd *occluders* där varje körning pågick i 3600 bildrutor. Totalt finns det 84240 polygoner i scenen. Som det går att se i grafen så ritas mindre polygoner ut när fler *occluders* i scenen används. Vissa perioder i körningarna renderas samma geometri vilket innebär att kurvorna överlappar vid de bildrutorna. Dock så är det aldrig fler polygoner som renderas i de körningar som använder fler *occluders* jämfört med körningarna som använder ett mindre antal *occluders*.



Figur 17 Hur många polygoner som renderades i Dust 2.

5.1.2 Scenen Peach's Castle

I det här delkapitlet presenteras resultatet av algoritmen för scenen Peach's Castle. Tabell 3 visar att overheaden för algoritmen ökar med mer utvalda *occluders* som förväntat. Algoritmen presterar inte mycket bättre i den här scenen som det går att se i tabell 4. När antalet utvalda *occluders* ökar så sjunker inte den genomsnittliga renderingstiden så mycket. Det är också liten skillnad på hur många polygoner som ritas ut i genomsnitt under körningen.

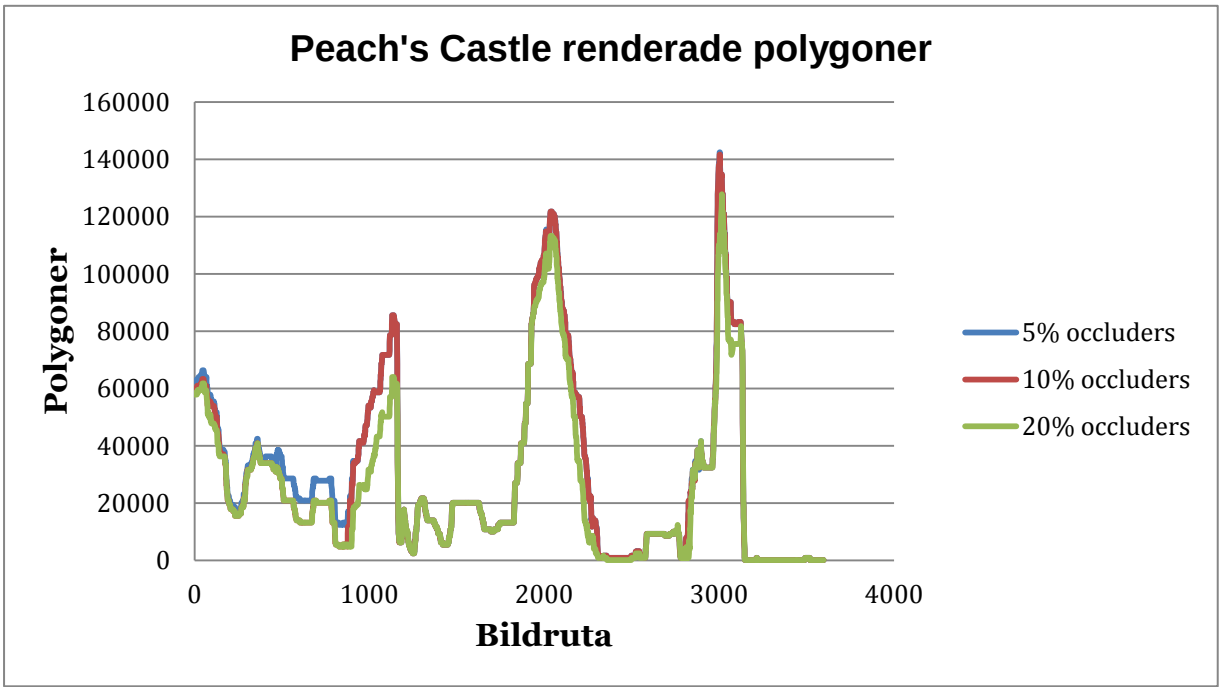
Tabell 3 Tabell som visar overhead av algoritmen i körningarna från scenen Peach's Castle.

Antal *occluders* i procent	**Antal utvalda *occluders***	**Genomsnittlig overhead [ms]**	**Minsta overhead [ms]**	**Högsta overhead [ms]**
5%	2	0.423	0.14	1.158
10%	5	0.486	0.136	1.837
20%	11	0.589	0.176	1.882

Tabell 4 Tabell som visar renderingstiderna i körningarna från scenen Peach's Castle.

Antal *occluders* i procent	**Genomsnittlig renderingstid [ms]**	**Minsta renderingstid [ms]**	**Högsta renderingstid [ms]**	**Renderade Polygoner %**
5%	0.534	0.166	1.128	10.78
10%	0.537	0.181	1.242	10.36
20%	0.503	0.178	1.025	9.19

Scenen som liknar Peach's Castle visar på att när det är en mer öppen scen så reduceras inte antalet renderade polygoner markant när antalet *occluders* ökar. I figur 18 visas antalet utritade polygoner för de tre körningarna där varje körning skett i totalt 3600 bildrutor. Scenen innehåller totalt 279564 polygoner. I de körningar som valt ut fler *occluders* renderas aldrig mer polygoner jämfört med de körningar som valt ut ett mindre antal *occluders*.



Figur 18 Hur många polygoner som renderades i Peach's Castle.

5.1.3 Scenen Lost Temple

I det här delkapitlet presenteras resultatet av algoritmen för scenen Lost Temple. Scenen innehåller totalt sett mindre potentiella *occluders*. Som det går att se i tabell 5 så ökar overheaden av algoritmen endast lite när fler *occluders* används. Tabell 6 visar att det inte är så stor skillnad i renderingstiderna och att antalet utritade polygoner är samma för varje körning.

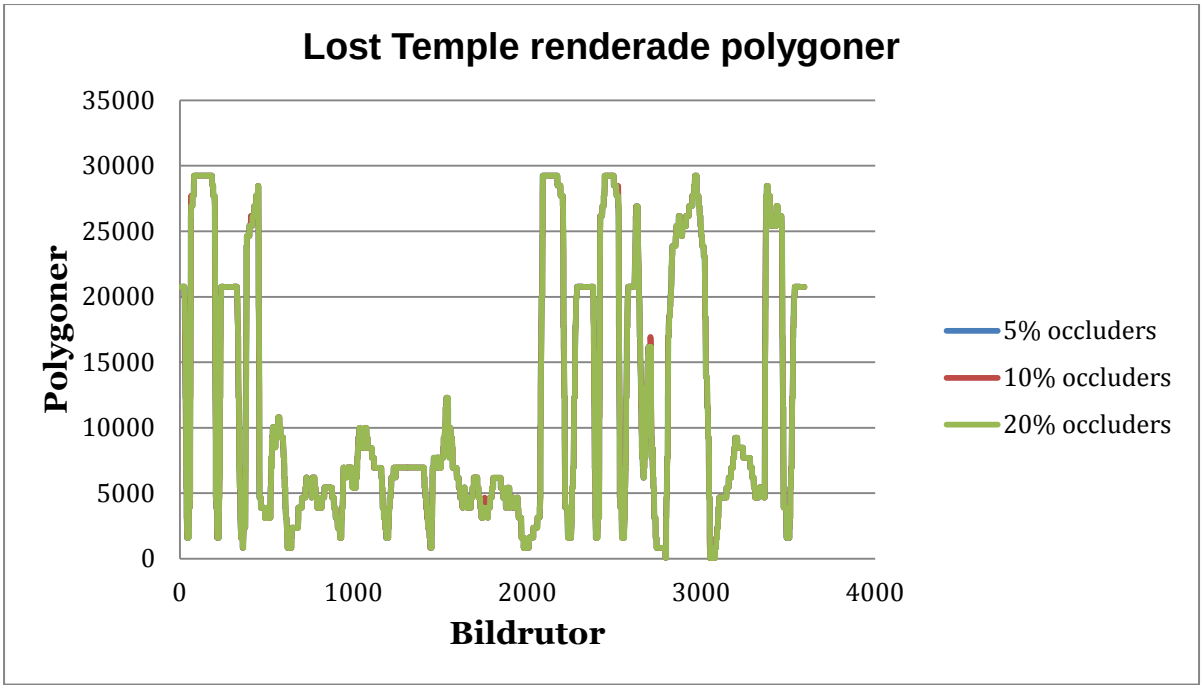
Tabell 5 Tabell som visar overhead av algoritmen i körningarna från scenen Lost Temple.

Antal *occluders* i procent	**Antal utvalda *occluders***	**Genomsnittlig overhead [ms]**	**Minsta overhead [ms]**	**Högsta overhead [ms]**
5%	1	0.248	0.098	0.926
10%	3	0.264	0.112	1.017
20%	6	0.280	0.107	0.998

Tabell 6 Tabell som visar renderingstiderna i körningarna från scenen Lost Temple.

Antal *occluders* i procent	**Genomsnittlig renderingstid [ms]**	**Minsta renderingstid [ms]**	**Högsta renderingstid [ms]**	**Renderade Polygoner %**
5%	0.442	0.177	1.461	5.58
10%	0.417	0.173	1.409	5.58
20%	0.450	0.154	1.177	5.58

I figur 19 så går det att se hur många polygoner som renderats i de tre körningar som skett för scenen Lost Temple. Varje körning pågick i totalt 3600 bildrutor. I scenen finns det totalt 210036 polygoner. Det är inte någon skillnad när antalet *occluders* förändras i en scen som Lost Temple. Under hela körningen så är kurvorna nästan identiska. Kameran är alltid i en position ovanifrån och det som åskådas kommer sannolikt inte att vara dolt av *occluders*.



Figur 19 Hur många polygoner som renderades i Lost Temple.

5.2 Presentation av körningarna med Unitys algoritmer

Eftersom det inte var möjligt att hämta ut samma information för Unitys algoritmer så presenteras resultatet av körningarna med Unitys algoritmer separat i det här delkapitlet. För att det skulle vara samma förutsättningar för det här projektets *occlusion culling*-algoritm och Unitys *occlusion culling*-algoritm så markerades alla potentiella *occluders* som statiska *occluders* i uppsättningen av scenen. Alla objekt i scenen markerades också som statiska *occludees* då alla objekt i de olika scenerna har möjlighet att vara dolda. Standardinställningar för Unitys *occlusion culling* användes också. Det är inte möjligt att se vilka *occluders* som väljs ut under körning eller hur många som används i varje bildruta så den informationen är inte med i resultatet. En annan skillnad för körningarna med Unitys algoritmer var att det inte gick att hämta ut hur många polygoner som ritades ut i varje bildruta så den informationen saknas också i presentationen.

Körningarna med Unitys algoritmer summeras genom att presentera tiden för overheaden av algoritmerna och tiden för renderingen. Varje körning har utförts i totalt 3600 bildrutor. För att ha samma förutsättningar som det här projektets algoritm inkluderades inte de 10 första bildrutorna även för Unitys körningar när det högsta värdet valdes ut för overheaden av algoritmerna och renderingstiderna.

5.2.1 Scenen Dust 2

I tabell 7 visas overheaden för körningarna med Unitys algoritmer i scenen Dust 2. Overheaden för Unitys *occlusion culling* är ungefär på samma nivå som det här projektets algoritms overhead. Som väntat är overheaden låg för Unitys *view frustum culling*. Det som är konstigt med resultatet är den högsta overheaden som gavs av Unitys *occlusion culling* körning. Vid närmare studie av resultatet visade det att fyra bildrutor hade en overhead på mer än 5 millisekunder. De fyra bildrutorna där overheaden var högre än 5 millisekunder låg utspritt under körningen och det gick inte att se något samband med andra tider.

Tiden för rendering i körningarna med Unitys algoritmer i scenen Dust 2 går att beskåda i tabell 8. Det som går att se här är att tiderna för rendering är högre än väntat för Unitys *occlusion culling*. Tiden för rendering med Unitys *view frustum culling* ligger på en förväntad nivå vilket var lite högre än resultatet med det här projektets *occlusion culling*-algoritm. Eftersom det är mer som borde ha varit dolt var förväntningen att det skulle gå fortare att rendera scenen med Unitys *occlusion culling* jämfört med att bara använda Unitys *view frustum culling*.

Tabell 7 Tabell som visar overhead av Unitys algoritmer i scenen Dust 2.

Unitys algoritmer	**Genomsnittlig overhead [ms]**	**Minsta overhead [ms]**	**Högsta overhead [ms]**
Occlusion culling	0.495	0.113	33.24
View frustum culling	0.117	0.042	0.285

Tabell 8 Tabell som visar renderingstiderna med Unitys algoritmer i scenen Dust 2.

Unitys algoritmer	**Genomsnittlig renderingstid [ms]**	**Minsta renderingstid [ms]**	**Högsta renderingstid [ms]**
Occlusion culling	1.12	0.181	2.602
View frustum culling	0.573	0.187	1.957

5.2.2 Scenen Peach's Castle

I tabell 9 visas overheaden för körningarna med Unitys algoritmer i scenen Peach's Castle. Även i den här scenen är resultatet för overheaden som väntat. Återigen så är det fyra bildrutor som har en uppmätt overhead på över 5 millisekunder. Eftersom det sker igen i körningen för den här scenen är det troligt att Unity utför beräkningar snarare än att det är något fel med hur scenen är uppsatt.

I tabell 10 går det att se tiden för rendering i körningarna med Unitys algoritmer i scenen Peach's Castle. Även för den här scenen är tiden för rendering oväntat hög för Unitys *occlusion culling*. Den minsta renderingstiden ligger på den nivå som förväntades men den genomsnittliga tiden är högre än både Unitys *view frustum culling* och det här projektets algoritms körningar. Tankarna förs igen till att Unity möjligtvis utför beräkningar i renderingssteget som inte hör till själva renderingen.

Tabell 9 Tabell som visar overhead av Unitys algoritmer i scenen Peach's Castle.

Unitys algoritmer	**Genomsnittlig overhead [ms]**	**Minsta overhead [ms]**	**Högsta overhead [ms]**
Occlusion culling	0.490	0.119	33.85
View frustum culling	0.117	0.05	0.326

Tabell 10 Tabell som visar renderingstiderna med Unitys algoritmer i scenen Peach's Castle.

Unitys algoritmer	**Genomsnittlig renderingstid [ms]**	**Minsta renderingstid [ms]**	**Högsta renderingstid [ms]**
Occlusion culling	1.283	0.182	2.371
View frustum culling	0.575	0.223	1.205

5.2.3 Scenen Lost Temple

I tabell 11 visas overheaden för körningarna med Unitys algoritmer i scenen Lost Temple. Som det går att se i resultatet är overheaden mycket mindre i den här scenen även för Unitys algoritmer. Det är inte något högt värde för högsta overheaden för Unitys *occlusion culling* som det var i de andra två scenerna. En anledning kan vara att det är mindre *occluders* i den här scenen eller så är det mindre beräkningar som behöver göras för att det inte finns många dolda objekt.

Tiderna för rendering i scenen Lost Temple med Unitys algoritmer visas i tabell 12. För den här scenen så är resultatet nästan som förväntat. Renderingstiden är inte mycket högre för Unitys *occlusion culling* vilket tyder på att det inte sker många extra beräkningar när det inte finns mycket i scenen som är dolt under körningen.

Tabell 11 Tabell som visar overhead av Unitys algoritmer i scenen Lost Temple.

Unitys algoritmer	**Genomsnittlig overhead [ms]**	**Minsta overhead [ms]**	**Högsta overhead [ms]**
Occlusion culling	0.176	0.071	0.648
View frustum culling	0.118	0.041	0.29

Tabell 12 Tabell som visar renderingstiderna med Unitys algoritmer i scenen Lost Temple.

Unitys algoritmer	**Genomsnittlig renderingstid [ms]**	**Minsta renderingstid [ms]**	**Högsta renderingstid [ms]**
Occlusion culling	0.480	0.191	0.992
V*iew frustum culling*	0.455	0.135	1.162

5.3 Analys

Resultatet av experimentet är intressant för det visar hur körningarna i de olika scenerna gav olika resultat från användandet av det här projektets algoritm. Används algoritmen i scener där det finns många bra *occluders* så kan en stor reducering i utritade polygoner noteras med en liten ökning i overheaden för algoritmen. Det visar också att en förändring i antal valda *occluders* kan ha en direkt påverkan på hur många polygoner som ritas ut.

I det här projektet har scener använts som efterliknar scener från olika spel. Det går inte säga om algoritmen presterar lika bra som den skulle göra i riktiga scener från spel. Det går däremot att se hur algoritmen påverkar vissa typer av scener mer än andra. Därför kan det vara en bra idé att tänka på att om *occlusion culling* kommer öka prestandan för spelet när det utvecklas eller om det endast ger en overhead att ha med *occlusion culling*.

Det här projektets algoritm är inte optimerad för större scener eller när det inte finns många objekt som är dolda. Det påvisas när algoritmen används i scenen Peach's Castle. Där används endast ett fåtal *occluders* men ändå är overheaden för algoritmen hög. Hade algoritmen optimerats och en rumsindelningsstruktur använts hade algoritmen kunnat fungera bättre. När inga objekt är dolda så testas alla synliga objekt mot alla synliga *occluders* vilket leder till hög overhead. I scenen Dust 2 finns det nästan alltid bra *occluders* som reducerar overheaden för algoritmen.

Resultatet visar på att det kan vara värt att lägga vikt i valet av *occluders* när en det finns en scen som kan dra nytta av det. I det här projektet påverkades Dust 2 mycket när antalet *occluders* förändrades. Samtidigt hade det kunnat vara bra att välja bort objekt som inte är lämpade som *occluders* tidigt i algoritmen för att reducera overheaden som ges. I det här projektet så valdes de objekt med störst volym ut som *occluders* i ett förbehandlingssteg. Under körning så behöver en *occluder* endast vara i kamerans frustum för att användas av algoritmen. Coorg och Teller (1997) och Bittner, Havran och Slavík (1998) använder sig av ett estimat för rymdvinkeln när de dynamiskt väljer ut *occluders* under körning. Det är möjligt att det hade kunnat reducera overheaden av det här projektets algoritm.

I scenen Lost Temple så påverkas inte antalet polygoner som renderas även om antalet *occluders* förändras. Det är möjligt att det är för få *occluders* som valts ut men samtidigt så är den scenens struktur och kameraåkning också annorlunda från de andra två scenerna. Aila och Miettinen (2004) skriver att om scenen ses från ovan så är *occlusion culling* näst intill värdelös och det går att utesluta 97 till 99 procent av alla potentiella *occluders*. Lost Temple scenen var också den enda scenen där alla renderingstider var jämbördiga.

Resultat av körningar med Unitys algoritmer och specifikt Unitys *occlusion culling*-algoritm var inte som förväntat. Det kan finnas fler anledningar till att renderingstiderna var högre än väntat men eftersom det inte går att granska deras lösning så diskuteras nedan några möjliga anledningar till det givna resultatet.

I manualen för Unity (2017) så står det att utvecklare ska lägga tanke till vilka objekt som markeras som *occluders* för att minska beräkningar som måste göras av systemet. Eftersom körningarna med Unitys algoritmer skulle vara jämbördiga med körningarna med det här projektets algoritm så användes samma objekt som potentiella *occluders* för båda algoritmerna. Det är möjligt att det är viktigare att se till att endast de objekt som tros kunna vara värdefulla *occluders* markeras som statiska *occluders* så att Unitys algoritm inte utför onödiga beräkningar på sämre lämpade *occluders*. Fler körningar med färre objekt markerade som statiska *occluders* hade varit en bra start för att se om det ger en högre eller en lägre renderingstid. En annan potentiell lösning kan vara att se över inställningarna för Unitys *occlusion culling* och testa andra värden för inställningarna då endast standardinställningarna användes i det här projektet. Det hade varit bra att testa om det förekommer höga värden för overheaden av Unitys algoritm i scener som Dust 2 och Peach's Castle när olika mängd objekt markeras som statiska *occluders*.

Det är också möjligt att Unitys rumsindelningsstruktur optimeras under renderingssteget och det bidrar till att Unitys *occlusion culling*-algoritm har en högre tid för rendering. Det är inte möjligt att se exakt vad som sker i de olika stegen i Unitys system vilket gör att det är möjligt att mer än rendering sker under renderingssteget.

Eftersom det går att se i Unitys scenvy att Unitys *occlusion culling* korrekt dolde de objekt som låg bakom *occluders* antogs det att objekt doldes även under körningarna. Då det inte går att göra en förfrågan om hur många polygoner som ritades ut under körningen är det svårt att se om det ritades ut lika få polygoner för Unitys algoritm som för det här projektets algoritm. Hade det varit möjligt att hämta hur många polygoner som ritades ut hade det varit möjligt att se om Unitys algoritm ritar ut det antal polygoner som den borde göra under en körning.

6 Slutsats

6.1 Sammanfattning

Problemet som formulerats i det här projektet var att undersöka effekten av att använda *occlusion culling* i olika spelscener och hur valet av *occluders* påverkar resultatet. I det här projektet har en *occlusion culling*-algoritm implementerats där det fanns möjlighet att välja hur stor mängd av en scens *occluders* som ska användas av algoritmen. Sedan har algoritmen studerats i tre olika spelscener för att se hur den presterar med olika förutsättningar.

De scener som skapats för projektet har satts upp på ett sätt som efterliknar scener från olika spel där strukturen och kameraåkningar varierar. Varje scen testades under tre körningar där antalet potentiella *occluders* skilde sig åt. För varje körning så sparades information som kan mätas. Antalet utritade polygoner under körningen sparades. Tiden det tar att rendera och overheaden som algoritmen ger sparades också ner. Resultatet av experimentet visar på att potentialen för *occlusion culling* varierar från scen till scen och det är inte garanterat att det förbättrar prestandan eller sänker antalet utritade polygoner.

6.2 Diskussion

För att kunna skapa komplexa scener där mycket ska hanteras på en gång är det viktigt att undersöka tekniker som kan reducera beräkningstider. Den forskning som skett för *occlusion culling* har fokuserat på scener där det är lämpligt att använda tekniken. Coorg och Teller (1997) skriver att deras algoritm utnyttjar stora *occluders* i stadsscener och inomhusscener och att det hade varit intressant att se påverkan i scener där det inte finns stora *occluders*. Det här projektet undersökte olika scener där det inte alltid fanns naturligt stora *occluders* att tillgå. Kameraåkningarna varierade också så att det fanns en variant som gör att *occlusion culling*-algoritmen aldrig blir relevant.

Att kunna välja en bra delmängd *occluders* är ett problem som diskuterats tidigare för att det direkt påverkar overheaden av *occlusion culling*-algoritmer. Om en algoritm skulle använda alla objekt i scenen som potentiella *occluders* skulle det ge en algoritm med $O(n^2)$ i tidskomplexitet. Därför är det viktigt att istället välja en delmängd av alla objekt. Bittner, Havran och Slavík (1998) skriver att det kan vara fördelaktigt att bestämma potentiella *occluders* när programmet startar.

En intressant aspekt av *occlusion culling* är nyttan den kan ge i komplexa stadsscener. Det kan underlätta simuleringar av städer där det finns mycket att rita ut. Det har skett mycket forskning för just stadsscener där det alltid var förmånligt att använda *occlusion culling* då det vanligtvis finns många bra *occluders*. Det här projektet har inte någon samhällelig nytta då valet av *occluders* i spelscener är ett specifikt problem som inte har något med samhället att göra. Några etiska aspekter är inte relevant då samhället inte påverkas i större grad av det här arbetet.

6.3 Framtida arbete

Det finns mycket som kan forskas om kring *occlusion culling*. Det arbete som gjorts i det här projektet kan också förbättras så att algoritmen är mer stabil och klarar av mer avancerade

scener. Hade en del mer arbete gjorts på algoritmen så att den klarar av att ta *occluders* med annan form än en kub så hade den kunnat användas i mer avancerade scener än de som finns i det här projektet. Möjligheten att testa algoritmen i fler spelscener hade också varit bra för att se att resultatet från experimentet som gjorts är korrekt. Det hade även varit intressant att implementera rumsindelningsstrukturer och se hur det påverkar overheaden av algoritmen.

Något som hade varit intressant hade varit att kunna räkna utritade polygoner i Unity för de körningar som gjordes med Unitys egna algoritmer. Då hade resultatet kunnat stärkas genom att se att korrekt antal polygoner renderades. Eftersom resultatet av körningarna med Unitys *occlusion culling*-algoritm visade att renderingstiderna tog längre tid så hade det varit bra att visa hur många polygoner som faktiskt ritades ut.

I det här projektet så undersöktes valet av *occluders* genom att sortera efter storleken på objektens volym och välja ut de största att använda som *occluders* i körningarna. En annan aspekt som hade varit intressant att undersöka är att dynamiskt välja *occluders* under körning med ett estimat av rymdvinkeln som flera tidigare projekt skrivit om. Att undersöka olika värden på estimatet för att se hur värdefulla *occluders* kan vara och inverkan det ger på tester.

Referenser

Aila, T. & Miettinen, V. (2004). dPVS: An Occlusion Culling System for Massive Dynamic Environments. *IEEE Computer Graphics and Applications*, *24*(2), 86-97.

Akenine-Möller, T. & Haines, E. (2002) *Real-time rendering* (2:a upplagan). Wellesley, MA: A K Peters, Ltd.

Assarsson, U. & Möller, T. (2000). Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools, 5*, 9-22.

Bacik, M. (2002). *Rendering the Great Outdoors: Fast Occlusion Culling for Outdoor Environments*. Tillgänglig på Internet: http://www.gamasutra.com/view/feature/131388/rendering_the_great_outdoors_fast_.php [Hämtad 17.03.27].

Bittner, J., Havran, V. & Slavík, P. (1998). *Hierarchical visibility culling with occlusion trees*. Proceedings of CGI'98, 22 June, 1998, Hannover, Germany.

Blizzard Entertainment (2010) *Starcraft 2: Legacy of the Void* (Version: 3.13.0.52910) [Datorprogram]. Blizzard Entertainment. Tillgänglig på Internet: http://eu.battle.net/sc2/en/

Chrysanthou, Y.L., Cohen-Or, D., Durand, F. & Silva, C.T. (2003). A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics, 9*, 412-431.

Cohen, J., Hoff, K., Hudson, T., Lin, M., Manocha, D. & Zhang, H. (1997) *Accelerated Occlusion Culling using Shadow Frusta* (s. 1-10). I: Boissonnat, J. D. (red.) Proceedings of the thirteenth annual symposium / Computational geometry (SCG '97), 4 juni, 1997, Nice, Frankrike.

Coorg, S. & Teller, S. (1996) *A Spatially and Temporally Coherent Object Space Visibility Algorithm*. Technical Report TM 546. Cambridge, MA: Laboratory for Computer Science vid Massachusetts Institute of Technology.

Coorg, S. & Teller, S. (1997). *Real-time occlusion culling for models with large occluders*. Symposium of Interactive 3D Graphics, 3 April, 1997, Providence RI, USA.

Courrèges, A. (2016). *DOOM (2016) - Graphics Study*. Tillgänglig på Internet: http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/ [Hämtad 17.02.17]

Manocha, D., Salomon, B. & Yoon, S. *Interactive view-dependent rendering with conservative occlusion culling in complex environments*. Proceedings of the 14th IEEE Visualization Conferece (VIS'03), 19-24 October, 2003, Seattle, Washington, USA.

Nintendo (1996) *Super Mario 64* (Version: 1.0) [Datorprogram]. Nintendo. Tillgänglig på Internet: http://www.nintendo.com/games/detail/super-mario-64-wii-u

Pantazopoulos, I. & Tzafestas, S. (2002). Occlusion culling algorithms: a comprehensive survey. *Journal of Intelligent and Robotic Systems, 35,* 123-156.

Power, J. & Rubino, C. (2008). Level Design Optimization Guidelines for Game Artists Using the Epic Games: Unreal Editor and Unreal Engine 2. *Computers in Entertainment*, *6*, No. 4, Article 55.

Umbra (Version 3.4) (2017) [Datorprogram] Helsingfors, Finland: Umbra. Tillgängligt på Internet: http://umbra3d.com/ [Hämtad 17.03.31].

Unity (Version 5.5.2) (2017) [Datorprogram] San Francisco, CA: Unity Technologies. Tillgängligt på Internet: http://unity3d.com/ [Hämtad 17.03.31]

Valve Corporation (2012) *Counter-Strike: Global Offensive* (Version: 2017.05.12) [Datorprogram]. Valve Corporation. Tillgänglig på Internet: http://blog.counter-strike.net/